A Survey on Thread-Level Speculation Techniques

ALVARO ESTEBANEZ, DIEGO R. LLANOS, and ARTURO GONZALEZ-ESCRIBANO,
Universidad de Valladolid

Thread-Level Speculation (TLS) is a promising technique that allows the parallel execution of sequential code without relying on a prior, compile-time-dependence analysis. In this work, we introduce the technique, present a taxonomy of TLS solutions, and summarize and put into perspective the most relevant advances in this field.

Categories and Subject Descriptors: F.1.2 [**Modes of Computation**]: Parallelism and Concurrency; D.1.3 [**Concurrent programming**]: Parallel programming

General Terms: Runtime parallelization

Additional Key Words and Phrases: Speculative multithreading, speculative runtime parallelization, thread-level data speculation, TLDS, optimistic parallelization, thread-level speculation, TLS

ACM Reference Format:
Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. 2016. A survey on thread-level speculation techniques. ACM Comput. Surv. 49, 2, Article 22 (June 2016), 39 pages.
DOI: http://dx.doi.org/10.1145/2938369

1. INTRODUCTION

Thread-Level Speculation (TLS), also called Speculative Parallelization (SP), or even Optimistic Parallelization, is a runtime technique that executes in parallel fragments of code that were originally intended to run sequentially. Instead of relying on compile-time analysis to identify independent parts of sequential code that can be run concurrently, TLS techniques optimistically assume that these parts can be executed in parallel by different threads. To ensure correctness, speculative threads should detect whether they have consumed a datum that was subsequently updated by a predeccessor thread, that is, a thread executing an earlier part of the code, according to sequential semantics. Such situations, called *dependence violations*, should be detected and rectified by hardware or software mechanisms, or a combination of both, to keep sequential semantics. If a dependence violation is detected, then a corrective action will take place, typically discarding the results calculated by the thread that has consumed the incorrect value and restarting it to be fed with the updated datum.

In this article we review the literature related to Thread-Level Speculation techniques, presenting a taxonomy that helps to better understand each proposed solution in its context. The article is organized as follows. Section 2 presents a global view of

This research was partially supported by MICINN (Spain) and the ERDF program of the European Union: HomProg-HetSys project (TIN2014-58876-P), CAPAP-H5 network (TIN2014-53522-REDT), and COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).
Authors' addresses: A. Estebanez, D. R. Llanos, and A. Gonzalez-Escribano, Departamento de Informática, Universidad de Valladolid, Paseo Belén 15, 47011, Valladolid, Spain; emails: {alvaro, diego, arturo}@infor.uva.es.

the problem, including a description of sources of speculation in the code, together with the main design choices that may arise while designing a TLS solution. Section 3 examines the first solutions that served as a base for the development of TLS systems. Section 4 details hardware-based approaches, where additional hardware is added to support speculation. Section 5 shows software-based proposals, which do not require additional hardware to monitor the parallel execution, at the cost of a certain performance loss. Section 6 describes other works that take advantage of TLS capabilities for different purposes. Section 7 cites some studies that have pointed out the theoretical and practical limits of the TLS paradigm. Finally, Section 8 concludes our article.

2. SOURCES OF TLS AND DESIGN CHOICES

In Torrellas [2011], an accurate summary of Thread-Level Speculation techniques is given, including a detailed description of the two main issues that any TLS system should solve: how to buffer and manage speculative states and how to detect and handle dependence violations. His analysis makes any effort to reproduce a summary of TLS characteristics here meaningless: We suggest that readers consult his work to better understand the fundamentals of the field and the management of side effects due to the use of thread-level speculation. In this section, we will briefly discuss where the main sources of speculation are, how TLS techniques can be classified, and which are the most important design choices that have to be faced to set up a TLS system.

2.1. Loops as a Source of Speculation

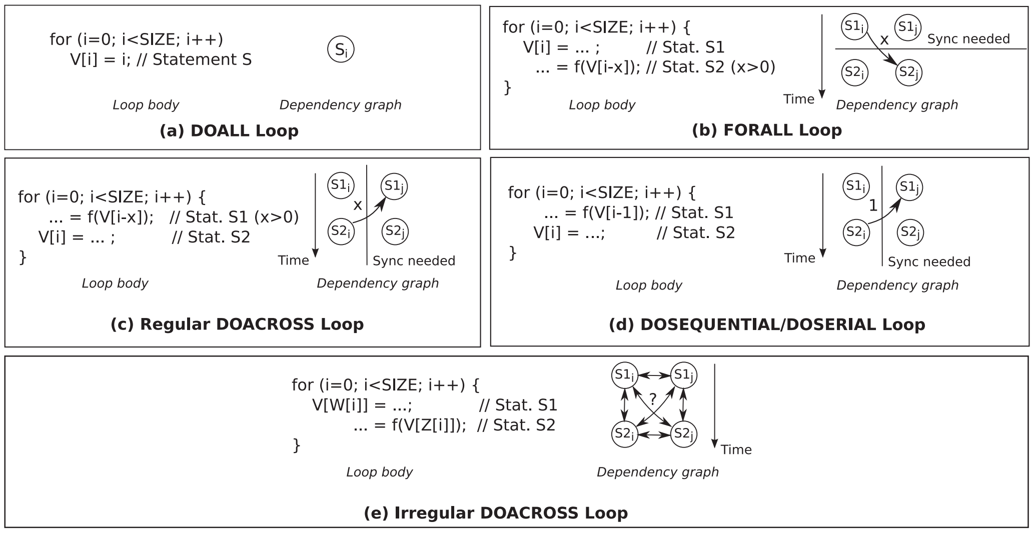
Due to how easy it is to distribute work among threads, loops are the most important source for TLS. The synthesis of loop-based speculation written by Rauchwerger [2011], who was also a pioneer in the field, accurately reflects the importance of loops as a source of speculation. Under TLS, loops are divided into blocks of iterations that are dispatched to be optimistically executed in parallel, while a monitor ensures that the execution follows sequential semantics. If this is not the case, then the monitor squashes offending threads, restarting them with the correct values. Otherwise, version data stored in the local speculative buffers are committed to the main copy. We will first briefly describe how data processed in one iteration may interact with calculations in different iterations, a situation known as data dependence.

There are three basic types of data dependencies among two fragments of code, namely *true*, *anti*, and *output* dependencies. In the following examples, let S_i and S_j be two statements, where S_i should be executed earlier than S_j according to sequential semantics.

—*True dependence:* Statement S_i writes into a location that is later read by S_j. These situations are also called Read After Write (RAW) conflicts, or *flow* dependencies.
—*Anti dependence:* Statement S_i reads a location that is later written by S_j. These situations are also called Write After Read (WAR) conflicts.
—*Output dependence:* Both statements S_i and S_j write into the same location. These situations are also called Write After Write (WAW) conflicts.

These definitions can be used to create a taxonomy of loops, according to the presence of data dependencies among their iterations. One of the first taxonomies was proposed by Polychronopoulos and Kuck [1987]. This work classified loops into three different types: *doall*, *forall*, and *doacross*.

—*Doall loops:* Loops that do not present any dependence among their iterations. Therefore, all iterations can be processed in parallel with no further checking [Tang and Yew 1986]. Figure 1(a) shows an example of this loop. Most of the current compilers can parallelize this kind of loops automatically.



```
for (i=0; i<SIZE; i++)
   V[i] = i; // Statement S
```

```
for (i=0; i<SIZE; i++) {
   V[i] = ... ;        // Stat. S1
      ... = f(V[i-x]); // Stat. S2 (x>0)
}
```

```
for (i=0; i<SIZE; i++) {
    ... = f(V[i-x]);   // Stat. S1 (x>0)
   V[i] = ... ;        // Stat. S2
}
```

```
for (i=0; i<SIZE; i++) {
      ... = f(V[i-1]); // Stat. S1
    V[i] = ...;        // Stat. S2
}
```

```
for (i=0; i<SIZE; i++) {
   V[W[i]] = ...;          // Stat. S1
          ... = f(V[Z[i]]); // Stat. S2
}
```

Loop body
Dependency graph
(a) DOALL Loop
Time
Sync needed
(b) FORALL Loop
(c) Regular DOACROSS Loop
(d) DOSEQUENTIAL/DOSERIAL Loop
(e) Irregular DOACROSS Loop
x
1
?

Fig. 1. Different types of loops according to the presence of data dependencies. The label in each edge represents the dependence distance. Data flows are represented by the arrow directions.

—*Forall loops:* Loop whose iterations may present true (that is, RAW) dependencies: Values produced by one iteration may be used in a subsequent iteration. An example is depicted in Figure 1(b). All iterations of a *forall* loop can be executed simultaneously if and only if all the statements that produce the value (S1 in the figure) have finished *before* the execution of any statement that consumes the value (S2 in the figure). If this behavior cannot be guaranteed, then a synchronization mechanism is needed.

—*Doacross loops:* Loops that may have cross-iteration (also known as WAR or backward) anti dependencies. Krothapalli and Sadayappan [1990] divides *doacross* loops into three categories:

  —*Regular doacross loops:* Loops whose antidependencies among iterations are dominated by a constant value x. Figure 1(c) shows an example. Regular doacross loops with x>1 can be parallelized by ensuring that the execution of the iterations involved in the dependence follows sequential semantics. If the value of x is known at compile time, then compilers are usually able to produce a parallel version of the loop.

  —*Dosequential or doserial loops:* A special type of *regular doacross* whose iterations depend on the previous one (that is, loops that have a dependence distance x=1). Figure 1(d) shows an example where the dependence is from the last statement of the body of the loop to the first statement. These loops have no parallelism at the iteration level.

  —*Irregular doacross loops:* Loops whose anti dependencies (also known as backward) among iterations are not known at compile time. Figure 1(e) shows an example. These loops are commonly called "irregular loops," and, in general, they cannot be parallelized safely at compile time.

Compile-time techniques can be used to generate parallel versions of *doall*, *forall*, and, when the dependence distance is known at compile time, *regular doacross* loops. Since TLS is a runtime technique, it can use the available information in all of the described loops, including *irregular doacross* loops. With respect to *dosequential* loops, a TLS system will also guarantee that the parallel execution will be correct, at the cost of squashing and re-starting iterations continuously to follow sequential semantics, thus degrading performance. The main application of TLS is in the parallel execution of *irregular doacross* loops when the total number of dependencies that appear at runtime is low.

2.2. Drawbacks of TLS

Although TLS can extract parallelism even from *irregular doacross loops*, it will likely be slower than a compile-time parallelization, if the latter can be applied. Sources of overhead in TLS include the cost associated to thread squash and restart due to data-dependence violations, speculative buffer overflows, load imbalance due to data locality issues, thread dispatch and commit, and inter-thread communications [Dou and Cintra 2004].

TLS overheads may not only lead to lower performance in terms of execution time but also to a greater energy consumption. This issue appears in software solutions, due to the energy cost associated to the execution of additional instructions to guarantee that sequential semantics are followed and to the wasted work carried out by squashed threads. Energy inefficiencies also appear in hardware approaches, due to the need of additional hardware structures in the cache hierarchy for data versioning, dependence checking, and its associated bus traffic [Renau et al. 2005]. We will return to this problem in Section 6.3.

2.3. A First Classification of TLS Techniques

According to Marcuello et al. [1998] and Kejariwal et al. [2006], there are three types of speculation techniques: (1) control speculation, (2) data dependence speculation, and (3) data values speculation (also called*value prediction*). These types are not disjoint, and their basis can be combined to achieve better results.

2.3.1. Control Speculation. Control speculation applies speculation to loops that include conditional sentences. Execution paths of each iteration are detected, mapping them to different threads. Jacobson et al. [1997b], Wallace et al. [1998], and Akkary and Driscoll [1998] combined control speculation with branch prediction. Puiggali et al. [2012] tried to predict the outcome of conditional branches without the need to know all the variables implied in the condition.

2.3.2. Data-Dependence Speculation. Data-dependence speculation is a technique suitable for the parallel execution of loops that may lead to inter-thread memory dependencies. Load operations from speculative variables (that is, variables whose use may lead to a dependence violation) usually return the most recent value for that variable, while speculative store operations search for the use of incorrect values in those threads, executing subsequent iterations according to sequential semantics. Many researchers have contributed to this solution: Please refer to Rauchwerger and Padua [1995], Franklin and Sohi [1996], Breach [1998], Marcuello et al. [1998], Cintra and Llanos [2003], and Tian et al. [2008].

2.3.3. Data Values Speculation. Data value speculation techniques, also known as *value prediction techniques*, predict at runtime the result of instructions before their execution. This approach is based on the idea that an accurate prediction may avoid a squash. For example, the work by Raman et al. [2008] describes a prediction-based TLS software that predicted values of the following iterations without specifying the iteration from where a value would be taken. The main disadvantage of these proposals is that, in general, for loops with irregular memory accesses and complex control flow, this solution does not obtain good predictions. Other works that use predictors are Sohi et al. [1995], Akkary and Driscoll [1998], Codrescu and Wills [1999a], Steffan et al. [2002], Cintra and Torrellas [2002], Prabhu and Olukotun [2003], Li et al. [2005], Tian et al. [2010a], Fan et al. [2012], and Gao2013.

2.4. Design Choices Overview

To be speculatively executed, the original code should be instrumented at compile or runtime to handle different operations, such as loading and storing of speculative data, performing commit operations if the speculative execution succeeds, and discarding incorrect work if it does not. The main design choices that should be faced in a TLS system are described in Yiapanis et al. [2013]. To implement a TLS system, a number of decisions should be made.[1]

2.4.1. Metadata Management. TLS approaches should manage some information in order to detect whether a dependence violation has occurred. Thus, each thread should know which memory addresses have been used, which operations have been done, and which thread has done each operation. All this information is collectively known as *metadata* [Yiapanis et al. 2013], and its management has two goals: preserving the information related to variables at risk of suffering violations, such as which thread has loaded, stored, or is locking a certain variable, and maintaining references about operations done by each thread, specifically, recording the variables loaded or written. The choice of the data structure to handle metadata may severely affect performance, depending on the relative costs of accessing and updating information during the parallel execution. An example of such a tradeoff can be found in Estebanez et al. [2014].

2.4.2. Version Management. When executing several consecutive fragments of sequential code in parallel, each thread usually maintains a version copy of the data structure that is accessed speculatively. This solution allows changes to these data to be performed locally, only storing these changes to a permanent place if the speculative execution of this thread proves successful. To do so, TLS systems require some additional storage to maintain the intermediate copies of each thread. There are two ways to manage these data:

—*Lazy Version Management*. In this case, a local copy of the exposed data is individually stored and managed. Therefore, when a load or store operation is performed, only the local version is changed. When a RAW dependence violation is detected, only local versions of threads in conflict have to be discarded, instead of modifying the reference version in memory.[2]
—The other approach, *Eager Version Management*, requires fewer resources, because the reference version in memory is modified. An additional buffer (called *undo log* in the literature) records old values and is used to restore original data in the case of a dependence violation.

Regarding version management, Garzarán et al. [2003, 2005] proposed a taxonomy to classify speculative systems according to the way of buffering the speculative versions of variables. They took into account the isolation of speculative thread states in each processor and how the new data versions produced by speculative threads is merged with the main memory.

2.4.3. Conflict Detection. Dependence violations can be checked with either a lazy or an eager approach: *Lazy Conflict Detection* avoids the need to check for conflicts on every access by delaying this task to a later stage before the commit operation. This

[1]Unless otherwise noted, the following discussion applies for both loop-level and block-level TLS systems.
[2]Note that WAW dependence violations can be avoided by a commit operation that follows sequential semantics. Regarding WAR dependencies, the use of local versions of exposed data avoids this problem.

solution implies the storage of the sequence of accesses to each speculative datum by different threads, in order to ensure that all accesses were performed following sequential semantics. Although this approach avoids time-consuming checks during the speculative execution, the amount of work that might be potentially discarded is much higher. A more strict approach, called *Eager Conflict Detection*, looks for potential dependence violations on every access. This design avoids performance losses produced by later checks by squashing and restarting threads as soon as a dependence violation is produced. However, the time devoted to checking each potential dependence violation is much higher, slowing down the parallel execution even when no dependence violations arise.

2.4.4. Scheduling of Iterations. To speculatively parallelize a loop, it should be partitioned into *chunks* (or *blocks*) of iterations to be assigned to different threads. Early approaches included a compile phase capable of classifying iterations into sets of independent iterations. Although iterations within a set should be executed in order, the sets should be executed sequentially, in order to avoid dependence violations. This *compile-time scheduling* solution came at the cost of performing a costly analysis that in many cases could not be carried out due to its complexity and/or the presence of potential dependence violations that depended on runtime information. In these cases, the simplest solution is to use *chunks of fixed size* [Kruskal and Weiss 1985]. The particular size chosen is an important design decision. The use of smaller chunks will reduce squashing costs, at the cost of a higher scheduling overhead. On the other hand, bigger chunks will increase the cost of thread squashing and may lead to load imbalance.

To mitigate these problems, variable chunk size strategies originally designed to achieve load balancing in parallel computations, such as Hummel et al. [1992] and Polychronopoulos and Kuck [1987], can also be used in speculative execution. Regarding the particular context of TLS, Llanos et al. [2007] proposed a *variable chunk size* for the speculative execution of randomized incremental algorithms, an important class of problems where the probability of a dependence violation decreases as execution proceeds. Their work uses smaller chunks for the first iterations, where randomized incremental algorithms present more dependence violations, then gradually increases the chunk size to reduce scheduling overheads, and, finally, reduces the size of the chunks again to achieve a better load balancing.

The use of chunk sizes that follows a predefined distribution, however, may not be the best solution. Speculative parallelization poses a more complex scheduling challenge than traditional parallelization, because, for irregular applications, both the number and the particular distribution of dependence violations are unknown before the loop is executed. Therefore, the idea of *changing the chunk size at runtime* depending on the number of squashes produced makes sense [Llanos et al. 2008]. Recently, Estebanez et al. [2015] proposed a method, called Moody Scheduling, that makes use of both the number of re-executions of the last chunks of iterations and their tendency (increasing, decreasing, stable) to figure out an appropriate chunk size for the following chunk to be scheduled.

2.4.5. Squashing Alternatives. If a RAW dependence violation is produced, then all data calculated by the offending thread (the one that have consumed the incorrect value) should be discarded. The mechanism chosen to do so is a design decision that severely affects performance. Some approaches just discard the threads that have consumed this particular, wrong value, and others discard the offending thread and all its successors. This leads to the following solution space, as described by Garcia-Yaguez et al. [2014]:

—*Stops parallel execution*: First solutions, such as Rauchwerger and Padua [1995], simply discard the entire speculation execution when a dependence violation was produced, and then restart the loop sequentially from the beginning. Due to their high cost in terms of execution time, these solutions only benefit loops that were indeed parallel.

—*Inclusive squashing*: This approach stops and restarts the first thread that have consumed the wrong value, together with all its successors, regardless of whether they have consumed any value from the offending thread. Due to its simplicity of implementation, this is the most used solution (see Cintra and Torrellas [2002], Cintra and Llanos [2003], Prabhu and Olukotun [2003], and Ceze et al. [2006]), although it may discard potentially useful work carried out by a successor that has not consumed any polluted data.

—*Exclusive squashing*: This approach squashes (a) the offending thread, (b) all successor threads that have consumed *any* value generated by him, and (c) all threads that have consumed*any* value produced by the aforementioned squashed threads. In other words, only successor threads that have not consumed any value that may be derived from the offending thread are allowed to survive. Note that this solution may discard threads that have consumed values from the offending thread that have no relationship with the value that triggered the dependence violation. Li et al. [2005] tried to implement this ideas in hardware. Colohan et al. [2006] also used this kind of squashing mechanism in the context of databases (where restarting a thread leads to big performance losses), and used sub-threads to check for squashed threads. Tian et al. [2010b] also proposed a solution that does not discard all the produced values, only a small part of them. Also, García-Yágüez et al. [2011] and Garcia-Yaguez et al. [2014] developed a software-only version of this idea, with the help of a list that stores which threads have consumed a value for a particular predecessor.

—*Perfect squashing*: Discards offending threads and those successors that have consumed *the incorrect value* or any value generated using it. Threads that have consumed correct values from the offending thread are not squashed. This is the approach that leads to fewer squashes. However, to keep track of the definition and use of each particular datum, an in-depth analysis should be performed. This operation seems to be too costly. For example, Akkary and Driscoll [1998] proposed a specific table to store dependencies, while Rotenberg et al. [1997] used a table that saved all intermediate values. Nevertheless, Tian et al. [2011] addressed this problem and concluded that this squash mechanism is not profitable.

The above discussion assumes that the data dependencies are handled at the data-element granularity level. Note that if the TLS system uses a granularity coarser than the data-element for speculative data, for example, at the cache level, *false conflicts* may appear, leading to unnecessary squashes of speculative threads.

The following section describes the ideas that led to modern TLS techniques.

3. PRECURSORS

One of the first approaches centered on the parallelization of loops that may present dependence violations was the one proposed by Knight [1986]. With the functional languages in mind, specifically the Multi-Lisp approach, Halstead [1985] introduced a hardware approach that allowed speculation through the use of two different caches, one dedicated to storing those values loaded from memory, and the other used to hold those values produced by the processor whose accuracy was not confirmed yet, thus using lazy version management (see Section 2.4.2). Midkiff and Padua [1987] described a solution to synchronize the concurrent execution of singly nested loops, while [Zhu and Yew 1987] described an algorithm to handle all types of loops described in Section 2.1. Aiken and Nicolau [1988] described another scheduling algorithm (see Section 2.4.4), which analyzed loops and obtained the optimal, dependence-free distribution, making use of compile-time analysis techniques (see Section 2.4.4). In those years, Baxter et al. [1989] performed research to extract some parallelism of *Doconsider* loops, a kind of regular *Doacross* loop (see Section 2.1), where iterations could be rearranged, in order to preserve dependence semantics and parallelize as many iterations as possible. They developed a compiler plugin that divided iterations into subsets of iterations that depend on each other to execute several independent subsets at the same time. Although this article was focused on programs whose dependencies are known at compile time, it also mentioned codes not schedulable at *start-time* [Mirchandaney and Saltz 1988; Saltz and Mirchandaney 1988], which are codes whose dependencies could only be extracted during their execution, and, therefore, a compile-time scheduling mechanism is not applicable.Krothapalli and Sadayappan [1988] explored a solution to remove anti and output dependencies (see Section 2.1). For that purpose, they performed a reference analysis, storing multiple copies of suspicious variables used in the loop. Later, Krothapalli and Sadayappan [1990] proposed a dynamic scheduler based on synchronism (see Section 2.4.4), that allowed *doacross* loops to be addressed with complex inter-iteration dependencies. Afterwards, Wolf and Lam [1991] used matrices to transform and parallelize loops in a general way, with the help of compile-time scheduling mechanisms capable of dealing with nested loops.

The idea of the use of a dynamic *inspector-executor* model appeared at that time. With this approach, an inspector loop checks for dependencies in a preliminary phase, and if no dependencies arise, a second phase executes the loop in parallel. Saltz et al. [1991] introduced this method in order to parallelize loops, showing that this technique allowed a significant performance improvement in loops with a big number of operations, where inspector phase time was not significant compared to the executor phase. However, none of these approaches parallelize loops with output dependencies. Chen et al. [1994] developed a software solution that reduced delays between processor communications and allowed the parallelization of loops with output dependencies. They reused some results during the execution, allowing the overlap of dependence iterations and the sharing of some information between inspector and executor phases.

4. HARDWARE-BASED APPROACHES

Several hardware implementations have been developed to support TLS, mainly through the addition of auxiliary registers to manage speculation. Even though most hardware approaches have some parts implemented in software, in this section, we will review both pure hardware-based and mixed implementations. There are mainly two ways to implement TLS on hardware (HTLS): Developing a chip from scratch, or customizing an existing chip. The modification of an existing chip led to the development of Simultaneous Multithreading (SMT) processors.[3]

This section is structured in three parts. The first describes the approaches that did not rely on any previously developed scheme; the second details those based on the SMT architecture; and the third depicts those that proposed CMP enhancements.

4.1. Pioneers

4.1.1. Multiscalar Paradigm. Sohi et al. [1995] developed the Multiscalar processor, one of the first and most important approaches that executed sequential code (called *tasks*) in parallel through speculation. The underlying idea was to perform some tasks in parallel with the use of a chip that included several processors, ensuring sequential semantics.

[3]Packirisamy et al. [2008] and Tang et al. [2005] compared SMT with Chip Multiprocessors (CMP) in the context of TLS, giving a perspective of performance, power, and thermal; Ungerer et al. [2003] described chips that support multithreading. However, a full description of these processors is beyond the scope of this article and will not be provided.

Parallelization was organized by using graphs of tasks. In this way, each processor received a task and executed it. Consistency was ensured with the help of additional control logic that synchronized the production of register values in predecessor tasks with the consumption in successor tasks. A hardware monitor also ensures correctness in speculative memory accesses. The execution of parallel tasks in each processor followed a fixed order, needed to ensure sequential semantics. To handle this, a ring of processors was proposed. If a processor used a wrong value from a predecessor, then its task was squashed and restarted (see Section 2.4.5). When each processor finished its execution, values were committed in the order imposed by the ring. As will be seen in Section 5.1.2, this idea was later used by several software-based TLS solutions to implement sliding-window mechanisms. The authors also suggested the use of a value predictor (see Section 2.3.3) to reduce squash overheads, and to improve load balance among processors in order to avoid wasting computational cycles, through the choice of an appropriate granularity. Sohi et al. [1995] affirmed that correctness of the operations could be ensured by different hardware implementations. A full description of one that supports the Multiscalar architecture can be found in Breach et al. [1994], Breach [1998], Franklin [1993], and Vijaykumar [1998]. Vijaykumar [1998] and Vijaykumar and Sohi [1998] also described efficient ways of choosing a good task division by using compile-time scheduling techniques (see Section 2.4.4).

Improvements in the Storage of Speculative Values. Several solutions tried to reduce overheads with the use of lazy version management (see Section 2.4.2) [Franklin and Sohi 1996; Gopal et al. 1998] describe several methods to support different data versions produced during speculative execution, through the use of hardware with the Multiscalar architecture. Franklin and Sohi [1996] proposed ARB, an Address Resolution Buffer used by all processors. This solution introduced some overheads due to the traffic caused by the simultaneous accesses to the ARB. Gopal et al. [1998] proposed a Speculative Versioning Cache (SVC), intended to overcome the limitations of ARB by assigning a different cache to each processor. Jacobson et al. [1997a] studied different branch prediction techniques for control speculation (see Section 2.3.1): an automata-based predictor, a prediction based on the history, and an address predictor for jumps and indirect calls.

4.1.2. The Trace Processor.

Rotenberg et al. [1997] developed an architecture based on the parallel execution of traces. Unlike the tasks used in the Multiscalar paradigm, which were obtained by the compiler dividing the sequential program, a *trace* is a dynamic sequence of instructions that are built as the program executes and stored in a so-called trace cache [Rotenberg et al. 1996]. This proposal consisted of a processor composed of different processing elements, each having the organization of a small-scale superscalar processor, with enough space to hold an entire trace and enough functional units and register files. Instructions were executed in parallel, while inter-trace dependencies were speculated with the use of value predictors.

Improvements to Trace. Patel et al. [1998] devised a way to reduce the size of traces and a modification of branches with the aim of making them more predictable. Black et al. [1999] modified the original Trace approach, managing traces as series of pointers to basic blocks stored in cache. Rotenberg and Smith [1999] addressed the problem of control independence to better exploit the parallelism of this architecture, using control speculation (see Section 2.3.1) to structure codes into control-independent code blocks. Jacobson and Smith [2000] improved the instruction dispatching of trace caches through the construction of sets of traces before they were needed. Several years later, Padmanabha et al. [2013] proposed the use of super-traces to detect those parts of codes that had regular patterns to maximize energy efficiency.

4.1.3. Oplinger et al. Architecture. Oplinger et al. [1997] combined ideas from the Multiscalar paradigm, putting more emphasis on the hardware and the compiler. The result was a chip that contained some subsystem caches that supported speculation. They also developed a specific software adapted to the underlying hardware. This work also contained a study of several benchmarks under different metrics, namely the number of sequential code lines, *doall* loops, *doacross* loops, and so on. The goal of this research was to check the parallel behavior of the benchmarks used. This study was improved by Oplinger et al. [1999], with the aim of locating niches for speculation. The authors concluded that TLS should not only be applied in a single loop. Instead, they affirmed that using TLS in all loops and procedures of sequential applications would produce better results.

4.1.4. I-ACOMA. Krishnan and Torrellas [1998] implemented a clustered SMT architecture where the chip had several independent processing units, with each unit having the capability to perform simultaneous multithreading. The system supported the speculative execution of binaries, without source recompilation, by the use of software that identified potential threads in programs. Their efforts were mainly centered on loops. The basis of this idea was to use a binary annotator that added some notes into the executable file. Order between threads was implemented through a bit mask, and communications between threads were performed by either memory or registers, throughout an annotation phase in the case of registers, and in runtime otherwise. Dependence violations were identified with the help of a table called the *Memory Disambiguation Table (MDT)* (an idea based on ARB, see Franklin and Sohi [1996]). The MDT was stored in the L2 cache and contained copies of the different data used.

4.1.5. STAMPede. Steffan and Mowry [1998] and Steffan et al. [2000] proposed a chip multiprocessor architecture with TLS support called STAMPede, whose goals were twofold: first, to handle arbitrary memory access patterns[4] and, second, to provide a scalable paradigm that could be adapted to both SMT and CMP architectures. They suggested a system that implements speculation with the help of a cache memory consistency protocol.

Value Prediction. The same authors improved the communication cost of the STAMPede architecture in Steffan et al. [2002]. To do so, they implemented value prediction (see Section 2.3.3), with no additional cost of recovering if prediction failed (speculation mechanisms were used in that case). Their solution tried to predict those values that were going to be loaded but not stored in the same epoch. The authors affirmed that these values were likely to produce dependence violations. They also proposed what they called “silent store,” which avoided the unnecessary stores of values to variables with the same value, replacing these stores, which led to many dependence violations, with loads that compared values to check correctness.

Compiler Optimization. Steffan et al. [2005] extended the cache protocol described by Steffan et al. [2000], developing a cooperative approach between the software and hardware parts of the system. Their goal was to optimize the compiler and reduce the complexity of the hardware used. Moreover, they adapted the hardware support to improve the compatibility with the majority of processors and to be scalable to any machine size.

Cache Locality. Fung and Steffan [2006] used an extended version of STAMPede to study the inherent problems related to cache locality when TLS executions were

[4]The reason was that the authors affirmed that previous works could only use array references (contrary to the opinion of Krishnan and Torrellas [1999]).

performed. The authors started their study on the initial transition from sequential to parallel execution. Cache misses at that point (called startup misses) were not significant enough to adversely affect performance. Instead, they found that read-only cache misses were the most important part of misses in the execution of programs. Write-based sharing parameters and strided miss patterns were also addressed in this article. The other misses were not intended to be mitigated because, according to their study, they represent less than 10% of the execution times of the benchmarks considered.

4.1.6. SPT Approach. Li et al. [2005] described an architecture specifically designed to support scalar applications. This work also contains the design of a compiler developed specifically for this architecture.[5] The architecture developed combined hardware and software parts. The proposed hardware was composed of two cores: one to execute the main program thread and the other to execute speculative threads. Li et al. [2003] described a software value predictor (see Section 2.3.3) in connection with this implementation.

4.2. Simultaneous Multithreading (SMT)

Tullsen et al. [1996, 1998] proposed to combine features available in both superscalar and multithreaded architectures. The original SMT paradigm was designed to improve the use of superscalar hardware with small additional cost by overlapping multiple threads on a single, wide-issue processor in order to allow independent threads to use different functional units in the same cycle. SMT added thread tags to a single-thread architecture and maintained a hardware context for every simultaneous thread, including a general register file, PC register, and other state registers. Tullsen et al. [1998] can be viewed as the starting point of the main research in TLS with SMT.

4.2.1. Speculative Multithreaded Processor. Marcuello et al. [1998] proposed a speculative multithreaded processor that extended the instruction window in order to use speculation through hardware. These authors developed an architecture that controlled parallelism without the need for a strict thread control. This scheme was entirely hardware based, so it did not need any modifications to existing binaries. This solution automatically detected loops (details of this detection can be found in Tubella and Gonzalez [1998]) and launched their parallel execution by several threads distributed along a ring. Communications among threads were supported using broadcast messages. This solution also used data prediction techniques (see Section 2.3.3). With the use of lazy version management (see Section 2.4.2), misspeculations were detected by the comparison of local values and versions of cache lines at commit time.

4.2.2. Threaded Multi-Path Execution (TME). Wallace et al. [1998] enhanced the SMT processor with control speculation capabilities (see Section 2.3.1), through the use of a branch predictor. The procedure followed was the execution of all possible branches of a loop, while there were enough resources. Otherwise, the most likely branch to occur was predicted. Before speculatively executing a predicted branch, threads saved their execution context just in case dependence violations were produced, and they had to re-execute the branch with correct values.

4.2.3. Dynamic Multithreading Processor (DMP). Akkary and Driscoll [1998] developed DMT, an architecture that fetched, renamed, and dispatched instructions from different locations of the same program into a modified hierarchy of small instruction windows. Each thread (created by hardware when a loop or a branch were located) managed and executed its own window of instructions. All its partial results, which

[5]A more detailed description of this compiler can be found at Du et al. [2004].

were stored in additional buffers and handled with lazy version management, were accessible by the other threads in order to detect misspeculations. This approach also used a branch predictor (see Section 2.3.1) to mitigate data dependencies between threads.

4.2.4. Implicitly-Multithreaded Processors (IMT). Park et al. [2003] developed IMT, based on the Multiscalar approach by Sohi et al. [1995] but mapping threads on SMT. This approach took some ideas from TME and DMT. IMT mainly differed from TME in the way the former managed control dependencies among threads: Whereas TME speculated with multiple threads in case of branch mispredictions (as previously exposed), IMT created threads taking into account the predictions made by threads in execution, trying to follow the program order. Regarding DMT, IMT led to better results than DMT, thanks to prediction enhancements, and also to the elimination of the selective recovering from misspeculations that required several searches.

4.2.5. Packirisamy et al. Solution. Packirisamy et al. [2006] extended the SMT architecture to support TLS with the use of a modified L1 data cache. The new cache scheme consisted of the addition of a pointer to the thread which stored a value in the cache, and some speculative bits to save the loads and stores performed to variables. The number of these bits was similar to the number of available threads, in order to avoid overheads in the check for dependence violations. According to Wang et al. [2006], this solution was more useful than DMT [Akkary and Driscoll 1998] and IMT [Park et al. 2003] when dealing with big threads. In addition, this approach used a simpler hardware than the one proposed by Marcuello et al. [1998].

4.3. Chip Multiprocessor

CMPs have also been widely used in conjunction with TLS solutions. Olukotun et al. [1996] was one of the first works that described these processors. A CMP can be viewed as a group of single-thread processors integrated onto the same processor chip, in order to act as a team. Generally, processors in a CMP have their own L1 cache and share the second-level cache. Olukotun et al. [2007] described some implementations of CMP such as Piranha [Barroso et al. 2000] and Niagara [Kongetira et al. 2005]. Here, we will review those studies that used CMP in connection to TLS.

4.3.1. SMT on CMPs. Krishnan and Torrellas [1999] explained in more detail the ideas described by Krishnan and Torrellas [1998] and extended the approach to CMPs, the chips used in all the following improvements. Cintra et al. [2000] designed and evaluated a different CMP architecture, based on the use of the mentioned MDT, for scalable speculative parallelization. Their solution required a relatively simple hardware and was efficiently integrated with the cache coherence protocol of a conventional NUMA (Non-Uniform Memory Access) multiprocessor. This integration of speculative chip multiprocessors into scalable systems seemed to offer great potential. Cintra and Torrellas [2002] later presented a hardware subsystem that aimed to learn, predict, and solve dependence violations that could arise. They addressed the reduction of squashes using an improved version of the MDT that contained information about data managed by the threads in order to perform predictions and making use of inclusive squashing (see Section 2.4.5). Martínez and Torrellas [2002] also proposed using synchronism in speculative threads to speculate in active barriers or busy logs. To do so, they developed a hardware solution that added a bit per line and some logic to the cache, also giving support for register checkpointing. If a misspeculation was detected, then the offending thread was restarted from a synchronization point.

4.3.2. Hydra CMP. The Hydra chip multiprocessor was developed by Hammond et al. [1998, 2000]. Hydra integrated four MIPS-based processors and their primary caches on a single chip, together with a shared secondary cache. Its design was influenced by two previous CMP designs that supported TLS: the Multiscalar paradigm [Sohi et al. 1995] and STAMPede [Steffan and Mowry 1998] approaches, both described previously. According to Hydra's authors, it was an intermediate step between them: Hydra allowed bigger threads with less complex processors than Multiscalar, where the use of a ring of processors and hardware-based thread sequencing used imposed these limitations. With respect to STAMPede, Hydra proposed more complex write-back primary caches that did not need to be drained as each thread completed, avoiding burst of bus activity. Hydra was also influenced by the architecture developed by Oplinger et al. [1997]. Hydra required the user to mark loops that would be speculatively executed, and incorporated a specific compiler and a runtime system with some instructions to manage speculation, including those needed to both start and end speculative loops, fork calls, and manage dependence violations. Olukotun et al. [1999] claimed that applying speculation only in loops, instead of speculating on both loops and procedures, led to performance improvements in connection to Hydra. This research also proposed some code transformations, and a hardware implementation of checkpoints to recover from misspeculations, with the help of backup copies of registers.

Profiling and Code Transformations. Chen and Olukotun [2003a] added an additional hardware component to Hydra, called TEST, that carried out the analysis of sequential codes to extract potential speculative loops. This profiling was later used in a dynamic parallelization system that transformed Java programs for speculative execution. Chen and Olukotun [2003b] used this tool to develop Java Runtime Parallelizing Machine (JRPM), a system that could dynamically and automatically parallelize many Java applications. JRPM served as an integration of all the previous approaches developed. This solution, implemented with the Java Virtual Machine as the abstraction layer, allowed the programmer to avoid the need for manual changes in the code to be parallelized almost entirely.

Tutorials and Performance Analysis. Prabhu and Olukotun [2003] developed a tutorial to explain the main adjustments needed to change a code in order to improve the performance earned with TLS, using Hydra TLS hardware implementation as their test platform. Optimizations described went from code reorganization to using value prediction (see Section 2.3.3) and to performing adjustments to the algorithm or to applying speculative pipelining. In a later work, Prabhu and Olukotun [2005] performed an in-depth study of some applications of the Standard Performance Evaluation Corporation (SPEC) CPU2000 benchmark suite with respect to speculative parallelization. The study includes a description of the applications used and tips on their parallelization using some of the techniques described above.

4.3.3. Atlas Chip-Multiprocessor. Codrescu and Wills [1999a] and Codrescu et al. [2001] aimed to extract parallelism using single-chip multiprocessors by developing a new chip based on the combination of TLS and inter-thread data value prediction (see Section 2.3.3). Their solution assigned fragments of sequential code to processors until all of them were busy. This assignment was carried out by a control module that also performed value prediction. This development followed the principles exposed by Rotenberg et al. [1997] and Marcuello et al. [1998], being one of the first approaches whose main design factor was based on value prediction. The main contribution of this approach was the use of a better predictor than those used in previous TLS researches, called AMA. It was based on three tables of recent values, as described in Codrescu and Wills [1999b]. The processor also contained a branch predictor for control speculation (see Section 2.3.1), together with a *recovery queue*.

4.3.4. Out-of-Order Speculative Execution. Renau et al. [2005] enhanced CMP with hardware updates, developing the first approach that allowed out-of-order spawn in the context of TLS, that is, the execution of some instructions in an unpredictable order, far from the sequential order imposed by previous works. This research also included a compiler that could parallelize codes to be executed in either in-order or out-of-order modes. DMT [Akkary and Driscoll 1998] processors could also execute out-of-order fragments of code (tasks), using an SMT processor with different structures and mechanisms. Akkary et al. [2008] later defined Disjoint Out-of-order execution (DOE) processors, a model that targeted inter-task data communications latencies. It was composed by several DOE cores, organized in a ring, that shared an L1 data cache and a task dispatcher. A DOE core consisted of two threads, one to execute independent instructions, and the other responsible of executing the dependent instructions. In this way, when a thread that depended on a previous thread execution needed to wait for a datum, its core executed its independent instructions while a previous datum was produced.

4.3.5. Bulk. Ceze et al. [2006] addressed the problem of the complexity of the basic operations involved in TLS processes. To do this, they used a metadata management system (see Section 2.4.1) consisting of a kind of hash encoder (called a *signature*) that managed the addresses accessed by each speculative thread. Each signature was a set of addresses and allowed several addresses to be treated as if they were a single one. They enhanced the architecture with some hardware mechanisms that could efficiently operate with this hashed information. The model developed supported TLS and also transactional memory. In the TLS approach, they proposed to assign all the updated values of this thread to the nearest successors when it began its execution and to get these data earlier. In addition, they developed some operations, such as intersection, union, and so on, to manage and operate with signatures. The details of the compiler used to develop this system were described in Liu et al. [2006].

4.3.6. POSH. Liu et al. [2006] developed the POSH compiler, which divided subroutines and loops into tasks. Instead of relying entirely on compile-time scheduling mechanisms (see Section 2.4.4), they used a profiling tool to select those tasks that could benefit from parallelism and data prefetching techniques. It was implemented in a CMP with some additional TLS registers and supported a software value predictor similar to the solution developed by Li et al. [2003].

4.3.7. RASP. The Runtime Automatic Speculative Parallelization (RASP) was the approach presented by Hertzberg and Olukotun [2011]. This hardware-based model could speculate over binary codes directly. A translator analyzed programs running in $\times 86$ systems and translated them into RISC microcode. The translator was based on the DBT86 system developed by Hertzberg and Olukotun [2009], with some enhancements to speculatively parallelize loops and acquire feedback. The new system could locate loops and add some instructions to allow commitments and rollbacks with checkpoints, thus performing lazy conflict detection (see Section 2.4.3). The main contribution of this research was the inclusion of an automatic tool to speculatively parallelize binary codes over the existing hardware.

4.4. Related Techniques

4.4.1. Transactional Memory. Transactional memory (TM) [Herlihy et al. 1993] is another form of optimistic execution [Barreto et al. 2012]. The main difference between TM and TLS is that TM was designed for a number of different goals, such as encapsulating synchronization burdens (such as deadlock avoidance) of fine-grained lock-based programming and to reduce contention by the use of optimistic speculation-based techniques. On the contrary, TLS was primarily designed to parallelize sequential applications. Therefore, TM portions of code intended to run in parallel portions do not need to follow any sequential order, while TLS threads need to preserve sequential order. Transactions have been combined with TLS in two ways: using transactions to guarantee sequential semantics of loops and speculating over large transactions to be executed in parallel. Regarding the first approach, Guo et al. [2008] proposed *LogSPoTM*, a hardware-based solution that enhanced *LogTM*, a transactional memory system, to ensure sequential semantics and give support to TLS. LogSPoTM was mainly based on the integration of timestamps and arbitration policies to impose a thread order and preserve semantics. Deng et al. [2012] used this solution and, with the help of a hardware value predictor, improved their previous experimental results. Dai et al. [2011] also used LogSPoTM in their work. They looked to reduce squashes with the use of a chip that supports packet-switch, Network-on-Chip, thus applying some of the ideas of network theories to speculation. This work considered that thread data were packets and assigned higher priority to those belonging to predecessor threads. This solution helped to reduce both timeouts and the number of squashes.

Porter et al. [2009] also used TLS to improve the performance of their hardware-based transactional model architecture. There are other works that combine TM techiques with software-based TLS; we will review them in Section 5.3.3.

4.4.2. Speculative and Transactional Lock Elision. Locks are a useful technique to guarantee sequential semantics of parallel programs. Rajwar and Goodman [2001] extended this concept with the so-called Speculative Lock Elision, allowing the speculative execution of critical sections. To do so, this solution automatically replaced locks by optimistic hardware transactions, checking that no errors were produced. If transactions failed, then the system used the original lock. Some studies were conducted to evaluate transactional lock elision supports in different architectures, including [Dice et al. 2009; Cain et al. 2013; Afek et al. 2014]. Some commodity processors offer this support, including Intel's Haswell [Hammarlund and et al. 2014] and IBM's Power 8 [Cain et al. 2013].

4.4.3. Database Context. Colohan et al. [2005] demonstrated that database transactions could benefit from the use of TLS when a large transaction (with more than 7,500 instructions) should be executed. Moreover, they gave some guidelines to remove some of the frequent dependencies of the software in this context. The hardware used to do so was based on STAMPede (see Section 4.1.5). Later, Colohan et al. [2006] introduced subthreads to manage dependencies that occur in those large threads. Their sub-threads introduced checkpoints and stored intermediate values to enable the recovery of some of the calculations done in the speculative execution until a dependence violation was produced, with the use of a lazy conflict detection system (see Section 2.4.3) that only discarded the wrong part of the execution.

5. SOFTWARE-BASED APPROACHES

Software-based TLS systems implement techniques to guarantee the coherence of the optimistic parallel execution on conventional processors, without the need for dedicated functional units. Research in this field has been centered on reducing, whenever possible, the overheads in execution times due to the need to ensure consistency by software.

As we will see, first proposals usually executed the loop in parallel, and if a dependence violation was produced, then the work already carried out was discarded, and the loop was re-executed sequentially. Subsequent approaches performed partial commits, in order to take advantage of the work carried out before a dependence violation appears, and thus tried to minimize the number of squashed threads to those that had actually consumed a polluted value and its sucessors (a technique known as *inclusive squashing* (see Section 2.4.5).

Again, we will follow a historical perspective to describe the research in this field. We will first center our attention on those solutions where programmers should explicitly invoke runtime library functions and/or compiler support to manage speculative execution. Then, we will move to solutions that are based on higher-level programming abstractions. We will finish this discussion with some proposals related to TLS behavior and a brief review of some works that mixed TLS with other techniques.

5.1. Solutions Relying on Compile-Time and Runtime Support

First approaches required programmers to use different methods to explicitly invoke TLS mechanisms. The most representative ones are described below.

5.1.1. LRPD Test. We can place the origins of Software TLS (STLS) in the work carried out by Rauchwerger and Padua [1995, 1999], with their research in the parallelism of *doall* loops. They proposed the use of a test called LRPD to support the speculative parallelization of loops with some backtracking capabilities. This proposal re-executed the loop serially if the runtime test failed, a squashing solution that is simple to implement but with a huge cost in case of misspeculation (see Section 2.4.5). The proposal worked as follows: The target loop was first transformed through privatization (that is, making private copies of shared variables) and reduction parallelization (determining at compile-time that certain operations are indeed reductions and replacing them with a parallel algorithm), and then it was speculatively executed as a *doall* loop. During this parallel execution, the test stored the iteration number where shared variables were defined and/or used. After the parallel loop execution, a fully parallel data dependence test was applied over this information to ensure that the loop had no cross-iteration dependence. If the test failed, then the loop was sequentially re-executed. Otherwise, the parallel execution of the loop was considered successful. This approach had the disadvantage of detecting cross-iteration dependencies only after the end of the parallel execution, thus implying a heavy performance penalty. Gupta and Nim [1998] presented a more efficient method for speculative array privatization that did not require the computation to be rolled back when a particular variable was found to produce a dependence violation. To do so, they presented a technique that allowed the early detection of loop-carried dependencies and another that detected parallelization hazards immediately after they were produced. In addition, they proposed a set of new runtime tests for speculative parallelization of loops that defied parallelization methods based solely on static analysis. Dang et al. [2002] developed a technique to extract the maximum available parallelism for loops that were known to present some dependencies. This solution presented an evolution of the LRPD test, called Recursive LRPD (R-LRPD). The basic idea was to transform a partially parallel loop into a sequence of fully parallel loops. At each stage, this proposal speculatively executed all remaining iterations in parallel and the R-LRPD test was applied to detect the potential dependencies.

5.1.2. Software Versions of Hardware Solutions. Rundberg and Stenström [2000] applied many of the ideas of hardware-based speculative architectures in software. First, name dependencies were solved by dynamically renaming data at runtime. Second, the overhead of restoring the original situation after a misspeculation was greatly reduced by reducing the amount of data to commit and by supporting parallel implementations of the commit phase. Third, some anti data-dependence violations were avoided by supporting lazy version management without the need to enforce synchronizations between a pair of conflicting threads. Fourth, true data-dependence violations were detected when they happened, which reduces the cost of misspeculations. To do so, each instruction on speculative data was augmented with a checking code that detects data-dependence violations dynamically, that is, using eager conflict detection (Section 2.4.3). Finally, it committed data following sequential semantics.

Cintra and Llanos [2003, 2005] developed a different scheme based on an aggressive sliding window. It checks for data-dependence violations on every speculative store, while avoiding synchronization whenever possible. The sliding window used consisted of an array of slots that store the status of each running thread and pointers to their own version of the speculative data. Commits were carried out in order from the non-speculative thread. Each time a commit operation was finished, the sliding window advanced one position, allowing a new, most-speculative thread to start. This solution used lazy version management, eager conflict detection, and inclusive squashing. More recently, Estebanez et al. [2016] improved this solution with a different implementation that supported the speculative access to dynamically allocated data structures and support for the use of pointer arithmetic. This solution used a sophisticated metadata management (see Section 2.4.1) with the help of hash tables to reduce the time needed to find the most up-to-date version of a datum (see Section 2.3.2), a problem also described in Tian et al. [2010b].

5.1.3. Based on Master/Slave Paradigm. Zilles and Sohi [2002] introduced the Master/Slave speculative parallelism, a new kind of speculation whose basics were the use of a master thread and some slaves that performed the task assigned by their master. The main idea of this technique was to divide at compile time the program into tasks that would be carried out by the slaves, while the master thread predicted the values that would be produced by each task and continued with the execution of the code without waiting for their results. This approximation needed to check all the values produced by slaves after the execution of a task with respect to the values predicted by the master. If both were equal, then the master's prediction had been successful; on the other hand, a misspeculation had been detected. In this case, the work incorrectly carried out by the master and all slaves since the last checkpoint needed to be discarded and re-executed.

5.1.4. Automatic Thread Extraction. Ottoni et al. [2005] proposed an automatic approach for thread extraction. The system, called DSWP, exploited the fine-grained pipeline parallelism of many applications to extract long-running, concurrently executing threads. Their results showed significant improvements when executing these applications on a dual-core CMP.

5.1.5. Complementing Compile-Time Techniques for Auto-Parallelization. Tournavitis et al. [2009] proposed the use of profile-driven parallelism detection to augment the number of loops that may considered safe to parallelize, relying on the user for final approval. This work also uses machine-learning techniques to take better mapping decisions for different target architectures.

5.1.6. Other Solutions: SpLIP, MiniTLS, and Later. Oancea et al. [2009] developed *SpLIP*, a speculative tool centered on decreasing overheads of speculative operations of previous approaches, implementing non-locking operations where possible, making use of hash functions for metadata management (see Section 2.4.1) and relying on versions of data instead of rollbacks (see Section 2.4.2). Yiapanis et al. [2013] introduced a new structure that reduced memory overheads of classical approaches based on the idea of mapping every user-accessed address into an array of integers using a hash function. The authors implemented this compact data structure in two approaches, namely *MiniTLS* and *Later*. The main characteristic of *MiniTLS* was that threads updated memory locations in-place and also that all operations followed fast and optimistic design patterns. As in SpLIP, hash functions were used for metadata management. However, this approach used rollback mechanisms instead of version management, because speculative threads modified values directly, possibly producing errors that needed to be handled. This solution is similar to SpLIP, so both were compared in this work. *Later* followed a different design, implementing a lazy version management of values, together with pessimistic design patterns in its operations. The structure used differed slightly, but it was based on the same operations and patterns. This approach also introduced a combination of inspector-executor techniques (described in Section 3) and the LRPD test (described in Section 5.1.1), implementing the new solution on them.

5.1.7. TLS Compiler and Runtime for Distributed Systems. Kim et al. [2012] present an automatic speculative parallelization system for clusters, composed of a parallelizing compiler and a speculative runtime that minimizes the overheads due to validations, through the use of lazy version management and conflict detection. Other STLS runtime solutions for distributed enviroments are covered in Section 5.4.

5.1.8. TLS for Web Applications. Martinsen et al. [2013] used a speculative mechanism in the context of web browsing. To do so, they implemented their software by means of the Squirrelfish JavaScript environment, which enabled the parallel execution of Javascript functions. They modified the Squirrelfish interpreter to enable each instance of the interpreter to be executed as a thread, while executing as many instances as functions. The used variables were maintained in a special vector that showed modified values to detect dependence violations. The use of TLS in this context allowed these authors to achieve noticeable speedups.

5.1.9. Apollo. Jimborean et al. [2012a], Jimborean [2012], and Jimborean et al. [2013] introduced a TLS framework specifically designed to speculatively execute nested loops by using features of the polyhedral model [Ancourt and Irigoin 1991] to dynamically transform code into a more optimized version that led to higher speedups. First, a compiler [Jimborean et al. 2012b] generated skeletons[6] that were the basis of executions, due to their ability to produce different code versions that could be selected at runtime. Then, a dynamic part was responsible for (a) representing memory accesses as predicting linear functions of the loop indices, with the help of interpolating functions, (b) performing dynamic dependence analysis and transformation selection, (c) instantiating the parallel skeleton code, and (d) guiding the execution. The execution was based on profiling the code several times during the execution in order to choose the polyhedral transformations that could better speed up the execution. The detection of dependence violations(see Section 2.3.2) was done at three levels: Basic scalars, memory accesses, and loop bounds. This framework led the authors to parallelize some benchmarks that had not been parallelized before due to dependence management hurdles.

5.1.10. HVD-TLS. Fan et al. [2012] developed a software-based speculative framework that improved classical TLS mechanisms by the development of new techniques to improve value prediction (see Section 2.3.3), value checking, dynamic task partition, and scheduling (see Section 2.4.4). Predictions performed were done using several predictors based on the original value of the variables in conflict. Such predictions used a predictor table that also maintained the number of correct predictions. Values were checked by the main thread to prevent committing unmodified values, a situation repeated many times according to the authors. Also, this system allowed different levels of granularity to be assigned at runtime, following a linear scheme or a heuristic scheme, where the system monitored the execution and changed the granularity accordingly.

[6]Skeletons [Darlington et al. 1993] are a set of high-order parallel forms intended to be used as basic building blocks for parallel implementations. They include program transformations to ease portability between different systems.

5.2. Solutions Relying on Additional Programming Abstraction Layers

Our second set of software-based solutions eased the use of TLS by offering new, higher-level abstraction layers.

5.2.1. Fast Track. Kelsey et al. [2009] developed a system called *Fast Track*, where a programmer can install potentially unsafe optimized code while leaving the task of error checking and recovery to the underlying implementation. Specifically, their programming interface allowed users to suggest faster implementations based on partial knowledge of a program and its usage. Fast Track divided code into two branches, the fast track and the normal track, and programmers could change between both tracks when needed. Their implementation included both compile-time and runtime support. A compiler inserts function calls to ensure that the fast track produced the same result as the sequential execution. To protect runtime data, the system relied on the compiler to insert checking code that protects stack data. Regarding global and heap data, the system relied on the operating system to protect them by turning off write permissions for them in both tracks and installing custom page-fault handlers. These handlers first recorded which page had been modified in an access map and then re-enabled write permissions. The runtime support checked program correctness through the comparison of results at the end of the tracks. If both results were similar, then results were supposed to be correct. Otherwise, the fast track results were discarded. In this system, one processor was reserved to run the fast track, and the rest to the execution of normal tracks.

5.2.2. The Copy-or-Discard Model. Tian et al. [2008, 2009] proposed the Copy-or-Discard (CorD) execution model, in which the execution of parallel threads was separately managed by a non-speculative one. Speculative threads read values of the non-speculative thread and performed their computation. After that, speculative threads were committed in order. Then, results were checked by a non-speculative thread to preserve the semantics of the sequential order, using a lazy conflict detection (see Section 2.4.3). The commit operation was performed by the non-speculative thread through the CorD mechanism, which checked whether results were correct. In this case, results were copied to the non-speculative data. Otherwise, they were discarded at no additional cost, thanks to the use of version copies.

CorD and Dynamic Memory. The CorD approach did not give support to those applications whose speculative variables were dynamically allocated, so Tian et al. [2010b] enhanced CorD to be used with programs that had such dynamic data structures. The main problem of this approach was data traversing, because a dynamic structure could change their size during the execution. Pointers imposed another problem, since a speculative copy of a dynamic structure might have a pointer with an address to a non-speculative copy. In order to solve these problems, they proposed using a mapping table that translated addresses among speculative and non-speculative threads. They also included optimizations in the treatment of linked structures. Finally, Tian et al. [2010a] used a value predictor to improve the parallelization of programs with frequent and predictable cross-iteration dependencies (see Section 2.3.2).

Reducing Misspeculations. Tian et al. [2011] later tried to further reduce misspeculations. They proposed an approach intended to reuse almost all the correct calculations performed by a thread whose iterations had suffered dependence violations, instead of discarding all this information, as most approaches did. To do so, they used a partial speculative space in addition to the primary speculative space of each thread. This new space maintained the first read values of a speculative variable. If a misspeculation was found, then only the successor spaces of the offending space were squashed (see Section 2.4.5). This approach led to better performance and to a reduction in the number of dependence violations, due to lower recovery times.

5.2.3. TLS Based on the Use of Compile-Time Directives. Bhowmik and Franklin [2002] described a compiler framework for TLS that allowed the parallelization of all instructions of a code, instead of only those that compose a loop. This feature specially benefited non-numerical applications with complex instructions. Codes were initially analyzed by the compiler and profiled to produce a control flow graph. It was then used to produce partitions that could be executed by multiple threads. Chen et al. [2003] also developed a compiler that focused on providing a quantitative analysis of codes with complex dependencies. Their aim was to give probabilities about the possible flows of the code and detect if a squash was likely to be produced.

Mitosis. Mitosis is a compiler framework developed by Quiñones et al. [2005] capable of deciding which fragments of code could be speculatively executed. To do so, the Mitosis compiler marked the beginning of a region whose outcome could be speculatively guessed with a so-called *spawning point (SP)*, and its end with the *control quasi-independent point (CQIP)* mark. When the sequential execution reached the SP, a speculative thread was launched. This thread predicted the possible values of the outcome of the parallel region and used them to start the speculative execution of the code from the CQIP. Meanwhile, the non-speculative thread continued its execution. If no errors were produced, then speculative threads were committed; otherwise, they were discarded. The choice of these *spawning points* was a key part of the work. To do so, marks were chosen with the use of a synthetic trace. It selected the most suitable parts of codes to be speculatively executed regarding some requirements, such as the amount of workload of routines with respect to the total, or possible misspeculations.

Spice C. SpiceC was an approach proposed by Feng et al. [2011]. SpiceC implemented a number of directives that, when added to sequential code, eased parallel programming. Programmers did not need to be particularly careful about communications or dependencies, because this model supported *doall*, *doacross*, pipelining, and speculative parallelism. This solution also supported dynamic structures and pointer addresses. SpiceC threads had their own private space for data. A shared global space was used to store shared data. Threads' first accesses were referred to shared space and loaded to each local space, to where following accesses were redirected. When threads ended their executions, they checked for misspeculations and committed their data to the shared space if they were correct. Directives were similar to OpenMP's [Dagum and Menon 1998], so sequential programs only needed a few additional directives: a directive to suggest what kind of parallelism would be used and another to mark where commit operations had to take place.

Feng et al. [2012a] extended SpiceC with some additional directives to support Input/Output (I/O) operations within parallel loops. To the best of our knowledge, this was the first approach that addressed the parallelization of this kind of code through TLS. The main idea behind this research was to break the cross-iteration dependencies caused by I/O operations (see Section 2.3.2) modifying the original code. To parallelize input operations, this approach calculated file pointers before entering the loop to be used in each iteration. File pointer copies were created on demand by the iterations that used them. Regarding output operations, they required the use of some additional buffers, in order to store intermediate outputs produced by each thread. Each output value was stored in the corresponding thread buffer and flushed at the end of each iteration following sequential semantics. Feng et al. [2012b] also augmented SpiceC directives to parallelize loops with dynamically linked data structures. This work tried to manage different data partitions of loops using the same code, addressing the problem of codes where multiple threads managed several data partitions.

ATLaS. Aldea et al. [2014, 2016] developed a Gnu C Compiler (GCC) plugin to add loop-based TLS support to OpenMP. Their proposal include the development of a new OpenMP *speculative* clause to be used in `for` loops [Aldea et al. 2012], which allowed programmers to declare all variables whose reads or writes may lead to dependence violations. The use of this clause guaranteed that all definitions and uses of speculative variables would follow sequential semantics. The ATLaS framework consisted of a GCC plug-in that gave support to the new *speculative* clause, and a runtime library that managed the speculative execution. The ATLaS runtime library is able to transparently support speculation over variables of any size and permits the use of pointer arithmetic. Its implementation offered version management; eager conflict detection; fixed, dynamic, and adaptive chunk scheduling; and both inclusive and exclusive squashing. The internals of the runtime library that managed dependence violations were described in Estebanez et al. [2016]. The entire ATLaS framework can be freely downloaded from atlas.infor.uva.es.

5.2.4. The Galois Model. Kulkarni et al. [2007, 2009] introduced *Galois*, a system that supported complex pointer-based sets of elements in optimistic parallelism. They were centered on benchmarks that should get a subset of points from a big set, in order to obtain a solution to the problem. To do so, they introduced two constructs, called *optimistic iterators*, that could be added to object-oriented programming languages like Java: The *set iterator*, intended to execute a loop that processes in parallel an unordered set of elements, and the *ordered-set iterator*, which traverses in parallel partially ordered sets while ensuring sequential semantics. The consistency of data was implemented using locks. Moreover, to allow recovery from misspeculations, all operations had their corresponding inverse methods. With this purpose, an undo log was defined for each iteration. In order to manage all iterations, this solution defined a commit pool that contained data such as the state of iteration executions or the position of the log. It controlled the entire execution, deciding how iterations were assigned and committed, conflicts were solved, and so on.

Efficiency Improvement through Data Partitioning. After that, Kulkarni et al. [2008] introduced some mechanisms aimed to improve the efficiency of Galois by better exploiting locality of reference, reducing misspeculation, and producing a lower synchronization overhead. The mechanisms proposed include data partitioning, to assign elements of data structures to cores; data-centric assignment policy, to improve locality; replacing fine-grained synchronization on data structure elements by coarser-grained synchronization on data structure partitions; and over-decomposition of data, to assign several partitions to the same core, thus avoiding that a lock on a partition stalls the execution of that core.

Scheduling. Kulkarni et al. [2008] addressed the problem of scheduling (see Section 2.4.4), developing an additional framework to Galois. Although iterations could be executed in any order within their baseline scheduling policy, this work showed the inefficiencies associated to this behavior and proposed an improvement based on clustering (select a cluster of iterations), labelling (assign the selected clusters to cores), and ordering (order of the clusters to be executed) of iterations. Scheduling strategies for irregular applications in TLS were also addressed by Jo and Kulkarni [2010] with Galois. Their strategies went from "stealing" the work of overloaded processors by idle processors in the static assignment to using a centralized place as a warehouse for the extracted partitions.

A Profiler: ParaMeter. Kulkarni et al. [2009b] developed a tool to extract parallelism profiles from irregular applications. This method took abstract measures of the inherent parallelism of the different points of a code, showing instructions that could be executed concurrently. Although this tool has been used in the context of Galois, the authors affirmed that it is framework independent. Kulkarni et al. [2009a] also introduced a suite of benchmarks to test irregular applications with the use of TLS libraries, including those used in the mentioned articles.

5.2.5. Optimizing Irregular Applications. Méndez-Lojo et al. [2010] described three manual techniques to optimize irregular applications in order to improve their parallel execution. The first one was based on the idea of modifying codes in such a way that all read operations were done before any write operation. The second one, called "one-shot," was based on the detection of dependencies before the execution. If none were detected, then checks for them could be disabled, and code could be parallelized without locks. Finally, for those algorithms whose bottlenecks were located in the accesses to datasets (appropriate for the benchmarks tested by them, described in Kulkarni et al. [2009a]), they developed the "iteration coalescing" optimization. While in Galois there was a one-to-one correspondence between iterations and data elements to be processed (which was called *activities*), this iteration allowed a single iteration to grab multiple active elements from the set of data elements to be processed (called the *working set*), thus reducing the overhead associated to their access. Later, Prountzos et al. [2011] completed this work by automatizing the manual techniques described. They addressed again the overhead problems that emerged from optimistic parallelization, specifically, those related to conflict checking and undo actions. The center of this research was to reduce locks and rollbacks of the shared objects, using some inferred properties. In 2011, Kulkarni et al. [2011] analyzed whether the order used to launch methods affected execution times.

5.2.6. SEED. An approach for speculative loop execution that handled nested loops was recently proposed by Gao et al. [2013]. They developed and implemented Statically GrEEdy and Dynamically Adaptive (SEED), a tool that provided a runtime scheduler capable of adaptively selecting loops for parallelization in terms of their potential benefits by performing a cost-benefit analysis that took into account the input data. This tool was composed by two phases, one related to compilation time, and the other related to runtime. In the compiler phase, loops were selected, threads were exposed in order to be later created, and the resulting code was optimized using precomputation and software value prediction (see Section 2.3.3) to reduce misspeculations. At runtime, the basic TLS operations, such as thread spawning, dependence violations detection, and squashes, were carried out, together with the use of adaptive scheduling techniques (see Section 2.4.4) that were sensitive to input data.

5.3. TLS Mixed with Other Techniques

5.3.1. Helper Threads, Runahead, and Multi Path Execution. Xekalakis et al. [2009] proposed a model that combined different techniques such as TLS, helper threads, and runahead execution, in order to dynamically choose at runtime the most appropriate combination. The helper threads technique is based on the runtime generation of small threads (also called slices) to improve the efficiency of the main thread, for example, by resolving highly unpredictable branches and cache misses. By contrast, runahead execution was based on executing instructions in advance when a long latency operation was expected. Runahead threads either ignore or predict the outcomes of this long latency operation. Consequently, runahead threads would be faster than the others, and they would serve to predict cache misses to help TLS threads. In other words, these prefetched threads were essentially helper threads acting in a runahead mode to help the execution of main threads. The main difference of this technique with respect to helper threads was that the former did not require additional threads. Xekalakis and Cintra [2010] later combined TLS with MultiPath execution, a technique consisting in executing the two branches of hard-to-predict branches. The main idea behind this approach was to enhance processors with multiple-context execution to enable a fast way to discard erroneous data of wrong branches. The execution had normal TLS and MultiPath modes, depending on the number of occurrences of hard-to-predict branches. The previous combinations were more detailed, extended, and mixed in Xekalakis et al. [2012], where the authors described a system that applied TLS to loops. Other techniques were also proposed, such as the use of prefetched threads when delays were detected.

5.3.2. Continuous Speculation. Zhang et al. [2010] described continuous speculation, a technique whose main objective was to achieve full occupancy of processors. For that purpose, they used speculation techniques to achieve the parallelization of large sequential codes. Their solution used a sliding window and a group classification to ensure the correct order of the tasks. To get information about the possibly parallel regions of a sequential code, they used Behavior-Oriented Parallelization (BOP) [Ding et al. 2007; Ding 2011], a tool that analyzed the program behavior to parallelize it.

5.3.3. STLS and Transactional Memory. There were several works that made joint use of TLS and TM solutions. Mehrara et al. [2009] described *STMLite*, an STM implementation customized to facilitate profile-guided automatic loop parallelization, by supporting TLS using a simplified variant of STM. STMLite was specifically designed to reduce overheads of accesses to logs of variables used in transactions. Raman et al. [2010] proposed *SMTX*, a software system that generalized existing software TLS memory systems to support speculative pipelining schemes and was tuned for loop parallelization. It was specifically designed to exploit hardware MTX (multi-thread transactions) capabilities. Conceptually, an MTX provides a private memory that was initialized with the contents of committed memory at the time of creation of the MTX. Several threads could participate concurrently in the MTX by performing loads and stores in this private memory. At the end of the MTX, if no conflictes were detected, then the contents of the private memory were committed. Otherwise, the MTX was rolled back.Barreto et al. [2012] proposed unifying software transactional memory and STLS in *TLSTM*. They developed a software tool that improved the execution of each transaction of parallel programs through the use of TLS.

5.3.4. Software-Based Lock Elision. Roy et al. [2009] proposed a software version of the speculative lock elision proposed by Rajwar and Goodman [2001] (see Section 4.4.2) that was fully implemented in software. If a misspeculation was produced, then the system executed the original lock. Synchronization and privatization were implemented through special instrumentation for objects and through signals between threads implemented inside the Linux kernel.

5.4. STLS on Distributed-Memory Systems

There have been some efforts on applying TLS techniques on clusters of commodity servers. Kim et al. [2010] present a runtime monitor called Distributed Software Multi-threaded Transactional Memory (DSMTX) that allows the application of pipeline parallelism, multi-threaded transactions and TLS on distributed-memory environments. Koduru et al. [2013] described dyDSM, a distributed-shared memory abstraction to process large dynamic graphs that provides support for exploiting speculative parallelism. The balance between communication and computation in graph-based applications is studied in Charan Koduru et al. [2014], proposing a new runtime, called ABC,[2] that dynamically modified the configuration of the underlying DSM. Finally, Palmieri et al.

[2011] proposed a transactional replication protocol, named OSARE, built on top of an Optimistic Atomic Broadcast (OAB) service, that in turn was designed to speed up Atomic Broadcasts in distributed-memory systems. OSARE opportunistically processes transactions in multiple, speculative serialization orders to increment the likelihood that the final message ordering established by the OAB service matches one of the already-speculated serialization orders.

5.5. STLS Using GPUs

Nowadays, parallelism applied to GPUs is one of the most profitable research fields due to their large number of computing units. This characteristic makes them desirable to find ways to use TLS with these architectures. Liu et al. [2010] discussed how TLS could be correctly used in the context of Graphical Processing Unit (GPU) computation. Meanwhile, Diamos and Yalamanchili [2010] extended *Harmony*, a runtime for heterogeneous, many-core systems, to support speculation in GPUs. Samadi et al. [2012] introduced *Paragon*, a solution that combined CPU and GPU executions to achieve the best performance. Feng et al. [2012, 2014] proposed a framework to run loops speculatively in GPUs. The main idea of their solution was to divide the tasks that should be carried out by a speculative runtime framework into five categories and to assign some of them to CPUs and the others to GPUs. Scheduling, results committing, and misspeculation recoveries were assigned to CPUs, while computation and misspeculation checks were carried out in GPUs. In a more recent approach, Zhang et al. [2013] introduced a new library based on sliding windows that support TLS in GPUs. Classical solutions that were expected to have a better behavior with GPUs, such as hybrid dependence checking and the use of a parallel commit scheme, were adapted by these authors to their software.

5.6. Other Proposals

Finally, we will now describe other software-based approaches that use speculation in particular domains.

5.6.1. Finite-State Machine in TLS. Zhao et al. [2012, 2014] introduced the use of probabilistic analysis into the design of speculation schemes. In particular, they focused on applications that were based on Finite-State Machines. The authors affirmed that this type of application has the most prevalent dependencies of all the programs. They developed a probabilistic model to formulate the relationship between speculative executions and the properties of the target computation and inputs. Based on that formulation, they proposed two model-based speculation schemes that automatically customized themselves with the best configurations for a given Finite-State Machine and its inputs. Zhao and Shen [2015] presented a set of techniques to remove the need for offline training to collect probabilistic properties that help to reduce the probability of misspeculations. Instead, their techniques allowed probabilistic analysis to be performed on the fly.

5.6.2. Mixed-Model Universal Software-TLS (MUTLS). Cao and Verbrugge [2013] introduced a mixed model to fork threads in both in-order and out-of-order ways in a TLS library. Their work was based on the use of the Low Level Virtual Machine (LLVM) compiler framework [Lattner and Adve 2004], which allowed multiple source languages and target architectures through the use of an intermediate representation. MUTLS allowed threads to fork and join in different parts of the code and also implemented barriers to avoid some rollbacks. Functions annotated with fork and join points are transformed at compile time. For each one, a speculative version was generated that included helper functions for interaction with the TLS runtime library, the use of synchronization points, and the assignment of local buffers. Threads were managed by four modules: One dedicated to maintaining the status of speculative threads, two dedicated to manage local and global variables of speculative threads, and the last one used to managing other modules and interact with the LLVM speculator pass.

5.6.3. TLS to Decompress: SDM. Jang et al. [2013] described a TLS scheme specifically designed to be applied to decompression algorithms. Their approach was centered on the application of prediction techniques based on partial decompression and pattern matching to quickly identify block chunks that can be independently decompressed. The tool decompressed in parallel all the blocks identified.

6. OTHER STUDIES RELATED TO TLS

There are several works that used TLS for other purposes, such as improving manual parallelization or performing module-level speculation. Other studies include how the energy consumed by TLS proposals could be reduced. In this section, we will review some of them.

6.1. TLS as a Help to Manual Parallelization

In order to avoid making speculative codes that might be slower than the original sequential ones, some researchers have proposed techniques to predict overheads of speculative parallelization. For example, the work developed by Dou and Cintra [2004, 2007] contained a compiler pass that can be used to estimate the overheads and the expected resulting performance gains, if any. Ding et al. [2007] proposed a software-based TLS system to help in the manual parallelization of applications. The system required the programmer to mark "possibly parallel regions" in the application to be parallelized. The system relied on a "tournament" model, with different threads cooperating to execute the region speculatively, while an additional thread ran the same code sequentially. If a single dependence arose, then speculation failed entirely and the sequential execution results were used instead. Ke et al. [2011] improved that work with a system that relied on dependence hints provided by the programmer. This allowed explicit data communication between threads, thus reducing runtime dependence violations. Ioannou and Cintra [2011] studied the problem of taking advantage of future many-core architectures by complementing parallel programming at a coarse-grained level with hardware TLS support to launch fine-grained implicit speculative threads. Other authors have focused on providing assistance to those programmers that extract TLS from the applications. For example, Aldea et al. [2011] and Wu et al. [2008] developed tools that made a static and/or dynamic profile of the codes, returning information that allowed a decision to be made about which loop would be the best candidate to be speculatively parallelized. Prabhu et al. [2010] developed some directives and operations to facilitate programmers to make their own speculative programs. Chen et al. [2004] designed a dependence profiler to extract information from a code. Bhattacharyya [2012] also developed a similar tool that studied the profitability of TLS with the use of profiling. More recently, Bhattacharyya and Amaral [2013] used polyhedral analysis to detect dependencies of loops at compile time (see Section 2.3.2), stating that this analysis overcame the previous one.

6.2. Module-Level Speculation

Module-level speculation is the application of speculation in a module-based layer. Chen and Olukotun [1998] applied this technique to object-oriented Java programs. Warg and Stenström [2001] compared the use of object-oriented and imperative languages in the context of module-level parallelism, concluding that there were no significant differences between either approach. Their experimental results showed that the use of this method in modules with a low computational load would adversely affect the performance of applications. Warg and Stenström [2003] described a predictor that allowed the detection of whether a module had a low overhead. In this case, no threads were created to help him. Later, Warg and Stenström [2005] developed another prediction technique to detect when misspeculations would occur, in order to avoid executions that were expected to be squashed (see Section 2.4.5). History-based prediction is used to prevent speculative threads from being spawned when they are expected to cause misspeculations. Pickett and Verbrugge [2006] addressed the requirements that Java language imposed to implement this approach, also analyzing their costs. They developed a design based on the application of module-level speculation to support TLS at the level of Java bytecode. To do so, they introduced two new bytecodes, to fork and join speculative threads, in order to ensure their correctness. They also gave an implementation of the mentioned design, called the SableSpMT analysis framework (described in more depth in Pickett and Verbrugge [2005] and Pickett [2007]).

6.3. Energy Consumption

Since their origins, TLS was claimed to be energy inefficient. Co et al. [2006] analyzed the energy consumption of some approaches based on the Trace processors described in Section 4.1.2. Renau et al. [2005, 2006] searched for the main sources of energy consumption in TLS, giving some advice for energy saving. With the same goal, Xekalakis et al. [2010] proposed a power allocation scheme for TLS systems based on Dynamic Voltage and Frequency Scaling (DVFS) that took power from non-profitable threads that would need to be discarded and used it to speed up more useful ones. Li and Guo [2010] proposed two algorithms also based on DVFS. They proposed both static and dynamic assignment algorithms, achieving significant reductions in energy consumption. However, in the work carried out by these authors, energy savings came at the cost of lower performances. This topic was also addressed by Luo et al. [2013], which developed a system that analyzed speculative execution and managed resources in a way that decreased the energy needed.

6.4. Benchmarks for TLS

Many of the aforementioned studies shared the same benchmarks to give experimental results. We could highlight SPEC benchmarks [Dixit 1993; Phansalkar et al. 2005; Henning 2006, 2007], a benchmark suite to measure computing performance. Other benchmark suites frequently used are *Olden* [Rogers et al. 1995], a set of relatively small programs that perform a monolithic task with minimal user feedback, and *MiBench* [Guthaus et al. 2001], a set of programs to test embedded systems. *STAMP* [Cao Minh et al. 2008] is a benchmark suite designed for transactional memory applications that was also used by different TLS approaches. The *LLVM* compiler infrastructure [Lattner and Adve 2004] also offered some benchmarks. *Princeton Application Repository for Shared-Memory Computers* [Bienia 2011] is another benchmark suite composed of multithreaded programs, and Kulkarni et al. [2009a] described *Lonestar*, a suite of benchmarks of TLS specifically developed to test the Galois system, described in Section 5.2.4.

7. LIMITS TO TLS

A number of articles are mainly centered on the analysis of TLS performance and its limitations. Although Prabhu and Olukotun [2005] affirmed that significant parallelism could be extracted using TLS in SPEC2000 applications, Kejariwal et al. [2006] affirmed that it is very difficult to achieve a high level of performance through TLS with this benchmark. Kejariwal et al. [2007] performed an analysis of TLS using SPEC CPU2006 benchmarks and affirmed that the use of TLS with these benchmarks did not lead to significant benefits, with just a 1% improvement. However, as Ioannou et al.

[2010] and Packirisamy et al. [2009] noted, the former study only considered parallelism at the innermost loop level, while the parallelization of outer loops would lead to speedups. Later, Kejariwal et al. [2010b] briefly analyzed the performance of TLS at the module level (also called the graph level). They studied factors such as recursion, or I/O, which limits TLS applicability at this level. Also, Kejariwal et al. [2010a] proposed an analytical model based on conditional probability to gauge the suitability of nested TLS.

Islam et al. [2007a, 2007b] analyzed loop-level parallelism in embedded applications with and without TLS, concluding that TLS is useful for extracting the most possible parallelism from this kind of programs. Finally, Bhattacharyya [2013] studied the influence of input sets in dependencies of some TLS benchmarks. To do so, he had proven 57 benchmarks of SPEC2006, PolyBench/C, BioBenchmark, and NAS in the IBMs BlueGene/Q supercomputer. The author concluded that the input set did not noticeably change the dependence behavior in the loops of the benchmarks studied.

8. CONCLUSIONS

Thread-level speculation is an active field, thanks in part to the appealing idea that it may be possible to automatically extract the loop-level parallelism of sequential applications without a prior and costly dependence analysis. Most studies described in this work have shown that TLS techniques effectively lead to a speedup when used under certain conditions. However, this technique is highly sensitive to the actual number of dependence violations that appear at runtime. A second drawback of TLS techniques is their comparatively high costs in terms of energy consumption.

While more general TLS solutions are developed, speculative-based techniques will likely coexist with other solutions in the execution of irregular codes that are not analyzable by other means. Current trends are focused on avoiding, as much as possible, dependencies with better predictors; developing advanced squashing techniques; and the use of TLS in many-core systems such as GPUs or Intel Xeon Phi.

ACKNOWLEDGMENTS

This article is dedicated in loving memory of Dr. Agustín de Dios Hernández.

REFERENCES

Yehuda Afek, Amir Levy, and Adam Morrison. 2014. Software-improved hardware lock elision. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*. ACM, New York, NY, 212–221.

A. Aiken and A. Nicolau. 1988. Optimal loop parallelization. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI'88)*. ACM, New York, NY, USA, 308–317. DOI:http://dx.doi.org/10.1145/53990.54021

Haitham Akkary and Michael A. Driscoll. 1998. A dynamic multithreading processor. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 31)*. IEEE Computer Society Press, Los Alamitos, CA, 226–236. http://dl.acm.org/citation.cfm?id=290940.290988

Haitham Akkary, Komal Jothi, Renjith Retnamma, Satyanarayana Nekkalapu, Doug Hall, and Shahrokh Shahidzadeh. 2008. On the potential of latency tolerant execution in speculative multithreading. In *Proceedings of the 1st International Forum on Next-generation Multicore/Manycore Technologies (IFMT'08)*. ACM, New York, NY, Article 3, 10 pages. DOI:http://dx.doi.org/10.1145/1463768.1463772

Sergio Aldea, Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. 2014. A new GCC plugin-based compiler pass to add support for thread-level speculation into OpenMP. In *Proceedings of the 20th International Conference on Parallel Processing (Euro-par'14)*. Springer-Verlag, Berlin, 234–245.

Sergio Aldea, Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. 2016. An OpenMP extension that supports thread-level speculation. *IEEE Trans. Parallel Distrib. Syst.* 27, 1 (Jan. 2016), 78–91.

Sergio Aldea, Diego R. Llanos, and Arturo Gonzalez-Escribano. 2011. Towards a compiler framework for thread-level speculation. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*. IEEE Computer Society, Washington, DC, USA, 267–271. DOI:http://dx.doi.org/10.1109/PDP.2011.14

Sergio Aldea, Diego R. Llanos, and Arturo Gonzalez-Escribano. 2012. Support for thread-level speculation into OpenMP. In *International Workshop on OpenMP (IWOMP 2012)*, Vol. LNCS 7312. 275–278.

Corinne Ancourt and François Irigoin. 1991. Scanning polyhedra with DO loops. In *ACM Sigplan Notices*, Vol. 26. ACM, 39–50.

João Barreto, Aleksandar Dragojevic, Paulo Ferreira, Ricardo Filipe, and Rachid Guerraoui. 2012. Unifying thread-level speculation and transactional memory. In *Proceedings of the 13th International Middleware Conference (Middleware'12)*. Springer-Verlag New York, Inc., New York, NY, 187–207. http://dl.acm.org/citation.cfm?id=2442626.2442639

Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. 2000. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*. ACM, New York, NY, 282–293. DOI:http://dx.doi.org/10.1145/339647.339696

D. Baxter, R. Mirchandaney, and J. H. Saltz. 1989. Run-time parallelization and scheduling of loops. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'89)*. ACM, New York, NY, 303–312. DOI:http://dx.doi.org/10.1145/72935.72967

Arnamoy Bhattacharyya. 2012. Using combined profiling to decide when thread level speculation is profitable. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, New York, NY, 483–484. DOI:http://dx.doi.org/10.1145/2370816.2370908

Arnamoy Bhattacharyya. 2013. Do inputs matter? Using data-dependence profiling to evaluate thread level speculation in BG/Q. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*. IEEE Press, Piscataway, NJ, 401–402. http://dl.acm.org/citation.cfm?id=2523721.2523775

Arnamoy Bhattacharyya and José Nelson Amaral. 2013. Automatic speculative parallelization of loops using polyhedral dependence analysis. In *Proceedings of the First International Workshop on Code OptimiSation for MultI and Many Cores (COSMIC'13)*. ACM, New York, NY, Article 1, 9 pages. DOI:http://dx.doi.org/10.1145/2446920.2446921

Anasua Bhowmik and Manoj Franklin. 2002. A general compiler framework for speculative multithreading. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'02)*. ACM, New York, NY, 99–108. DOI:http://dx.doi.org/10.1145/564870.564885

Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

Bryan Black, Bohuslav Rychlik, and John Paul Shen. 1999. The block-based trace cache. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*. IEEE Computer Society, Washington, DC, 196–207. DOI:http://dx.doi.org/10.1145/300979.300996

Scott Elliott Breach. 1998. *Design and Evaluation of A Multiscalar Processor*. Ph.D. Dissertation. The University of Wisconsin—Madison. AAI9910432.

Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. 1994. The anatomy of the register file in a multiscalar processor. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO 27)*. ACM, New York, NY, 181–190. DOI:http://dx.doi.org/10.1145/192724.192750

Harold W. Cain and et al. 2013. Robust architectural support for transactional memory in the power architecture. *ACM SIGARCH Comp. Arch. News* 41, 3 (2013), 225–236.

Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. 2013. Robust architectural support for transactional memory in the power architecture. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, New York, NY, 225–236.

Zhen Cao and C. Verbrugge. 2013. Mixed model universal software thread-level speculation. In *Parallel Processing (ICPP), 2013 42nd International Conference on*. IEEE Computer Society, Washington, DC, 651–660. DOI:http://dx.doi.org/10.1109/ICPP.2013.79

Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE Computer Society, Washington, DC, 35–46. DOI:http://dx.doi.org/10.1109/IISWC.2008.4636089

Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. 2006. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA'06)*. IEEE Computer Society, Washington, DC, 227–238. DOI:http://dx.doi.org/10.1109/ISCA.2006.13

Sai Charan Koduru, Keval Vora, and Rajesh Gupta. 2014. ABC2: Adaptively balancing computation and communication in a DSM cluster of multicores for irregular applications. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, Los Alamitos, CA, 391–400.

Ding Kai Chen, Josep Torrellas, and Pen Chung Yew. 1994. An efficient algorithm for the run-time parallelization of DOACROSS loops. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing (Supercomputing'94)*. IEEE Computer Society Press, Los Alamitos, CA, 518–527. http://dl.acm.org/citation.cfm?id=602770.602857

M. K. Chen and K. Olukotun. 1998. Exploiting method-level parallelism in single-threaded java programs. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*. IEEE Computer Society, Washington, DC, 176–184. DOI:http://dx.doi.org/10.1109/PACT.1998.727190

Michael Chen and Kunle Olukotun. 2003a. TEST: A tracer for extracting speculative threads. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO'03)*. IEEE Computer Society, Washington, DC, 301–312. DOI:http://dx.doi.org/10.1109/CGO.2003.1191554

Michael K. Chen and Kunle Olukotun. 2003b. The Jrpm system for dynamically parallelizing java programs. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*. ACM, New York, NY, 434–446. DOI:http://dx.doi.org/10.1145/859618.859668

Peng-Sheng Chen, Ming-Yu Hung, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee. 2003. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*. ACM, New York, NY, 25–36. DOI:http://dx.doi.org/10.1145/781498.781502

Tong Chen, Jin Lin, Xiaoru Dai, Wei-Chung Hsu, and Pen-Chung Yew. 2004. Data dependence profiling for speculative optimizations. In *Compiler Construction*, Evelyn Duesterwald (Ed.). Lecture Notes in Computer Science, Vol. 2985. Springer, Berlin, 57–72. DOI:http://dx.doi.org/10.1007/978-3-540-24723-4_5

Marcelo Cintra and Diego R. Llanos. 2003. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*. ACM, New York, NY, 13–24. DOI:http://dx.doi.org/10.1145/781498.781501

Marcelo Cintra and Diego R. Llanos. 2005. Design space exploration of a software speculative parallelization scheme. *IEEE Trans. Paral. Distr. Syst.* 16, 6 (June 2005), 562–576.

Marcelo Cintra, José F. Martínez, and Josep Torrellas. 2000. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*. ACM, New York, NY, 13–24. DOI:http://dx.doi.org/10.1145/339647.363382

Marcelo Cintra and Josep Torrellas. 2002. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA'02)*. IEEE Computer Society, Washington, DC, 43. http://dl.acm.org/citation.cfm?id=874076.876479

Michele Co, Dee A. B. Weikle, and Kevin Skadron. 2006. Evaluating trace cache energy efficiency. *ACM Trans. Archit. Code Optim.* 3, 4 (Dec. 2006), 450–476. DOI:http://dx.doi.org/10.1145/1187976.1187980

L. Codrescu and D. S. Wills. 1999a. Architecture of the atlas chip-multiprocessor: Dynamically parallelizing irregular applications. In *Proceedings of the 1999 IEEE International Conference on Computer Design (ICCD'99)*. IEEE Computer Society, Washington, DC, 428–435. DOI:http://dx.doi.org/10.1109/ICCD.1999.808577

Lucian Codrescu and D. Scott Wills. 1999b. On dynamic speculative thread partitioning and the MEM-slicing algorithm. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*. IEEE Computer Society, Washington, DC, 40. http://dl.acm.org/citation.cfm?id=520793.825737

Lucian Codrescu, D. Scott Wills, and James Meindl. 2001. Architecture of the atlas chip-multiprocessor: Dynamically parallelizing irregular applications. *IEEE Trans. Comput.* 50, 1 (Jan. 2001), 67–82. DOI:http://dx.doi.org/10.1109/12.902753

Christopher B. Colohan, Anastassia Ailamaki, J. Gregory Steffan, and Todd C. Mowry. 2005. Optimistic intra-transaction parallelism on chip multiprocessors. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB'05)*. VLDB Endowment, Trondheim, Norway, 73–84. http://dl.acm.org/citation.cfm?id=1083592.1083604

Christopher B. Colohan, Anastassia Ailamaki, J. Gregory Steffan, and Todd C. Mowry. 2006. Tolerating dependencies between large speculative threads via sub-threads. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA'06)*. IEEE Computer Society, Washington, DC, 216–226. DOI:http://dx.doi.org/10.1109/ISCA.2006.43

Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.* 5, 1 (Jan. 1998), 46–55. DOI:http://dx.doi.org/10.1109/99.660313

Wenbo Dai, Hong An, Qi Li, Gongming Li, Bobin Deng, Shilei Wu, Xiaomei Li, and Yu Liu. 2011. A priority-aware NoC to reduce squashes in thread level speculation for chip multiprocessors. In *Proceedings of the 2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications (ISPA'11)*. IEEE Computer Society, Washington, DC, 87–92. DOI:http://dx.doi.org/10.1109/ISPA.2011.21

Francis Dang, Hoo Yu, and Lawrence Rauchwerger. 2002. The R-LRPD test: Speculative parallelization of partially parallel loops. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02)*. IEEE Computer Society, Washington, DC, 318–327. DOI:http://dx.doi.org/10.1109/IPDPS.2002.1015493

John Darlington, Anthony J. Field, Peter G. Harrison, Paul H. J. Kelly, David W. N. Sharp, Qian Wu, and Ronald L. While. 1993. Parallel programming using skeleton functions. In *PARLE'93 Parallel Architectures and Languages Europe*. Springer, Berlin, 146–160.

Bobin Deng, Hong An, Qi Li, Gongming Li, and Mengjie Mao. 2012. Value predicted LogSPoTM: Improve the parallesim of thread level system by using a value predictor. In *Proceedings of the 2012 IEEE/ACIS 11th International Conference on Computer and Information Science (ICIS'12)*. IEEE Computer Society, Washington, DC, 130–135. DOI:http://dx.doi.org/10.1109/ICIS.2012.116

G. Diamos and S. Yalamanchili. 2010. Speculative execution on multi-GPU systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE Computer Society, Washington, DC, 1–12. DOI:http://dx.doi.org/10.1109/IPDPS.2010.5470427

Dave Dice, Yossi Lev, Mark Moir, Dan Nussbaum, and Marek Olszewski. 2009. *Early Experience with a Commercial Hardware Transactional Memory Implementation*. Technical Report. Sun Microsystems, Inc.

Chen Ding. 2011. Parallel programming by hints. In *Proc. of the ACM International Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA'11)*. ACM, New York, NY, 13–14. DOI:http://dx.doi.org/10.1145/2048147.2048154

Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. 2007. Software behavior oriented parallelization. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI'07)*. ACM, New York, NY, 223–234. DOI:http://dx.doi.org/10.1145/1250734.1250760

Kaivalya M. Dixit. 1993. Overview of the SPEC Benchmarks. (1993).

Jialin Dou and Marcelo Cintra. 2004. Compiler estimation of load imbalance overhead in speculative parallelization. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*. IEEE Computer Society, Washington, DC, 203–214. DOI:http://dx.doi.org/10.1109/PACT.2004.12

Jialin Dou and Marcelo Cintra. 2007. A compiler cost model for speculative parallelization. *ACM Trans. Archit. Code Optim.* 4, 2, Article 12 (June 2007), 11 pages. DOI:http://dx.doi.org/10.1145/1250727.1250732

Zhao-Hui Du, Chu-Cheow Lim, Xiao-Feng Li, Chen Yang, Qingyu Zhao, and Tin-Fook Ngai. 2004. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI'04)*. ACM, New York, NY, 71–81. DOI:http://dx.doi.org/10.1145/996841.996852

Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. 2014. New data structures to handle speculative parallelization at runtime. In *Proceedings of the 7th International Symposium on High-level Parallel Programming and Applications (HLPP'14)*. ACM, New York, NY, 239–258.

Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. 2016. New data structures to handle speculative parallelization at runtime. *Int. J. Parallel Program.* 44, 3 (June 2016), 407–426.

Alvaro Estebanez, Diego R. Llanos, David Orden, and Belén Palop. 2015. Moody scheduling for speculative parallelization. In *Euro-Par 2015: Parallel Processing*. Springer, Berlin, 135–146.

Xu Fan, Shen Li, and Wang Zhiying. 2012. HVD-TLS: A novel framework of thread level speculation. In *Proceedings of the 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications (TRUSTCOM'12)*. IEEE Computer Society, Washington, DC, 1912–1917. DOI:http://dx.doi.org/10.1109/TrustCom.2012.176

Min Feng, Rajiv Gupta, and Laxmi N. Bhuyan. 2012. Speculative parallelization on GPGPUs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM, New York, NY,, 293–294. DOI:http://dx.doi.org/10.1145/2145816.2145860

Min Feng, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. Optimistic parallelism on GPUs. In *Languages and Compilers for Parallel Computing*. Springer, Berlin, 3–18.

Min Feng, Rajiv Gupta, and Yi Hu. 2011. SpiceC: Scalable parallelism via implicit copying and explicit commit. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. ACM, New York, NY, 69–80. DOI:http://dx.doi.org/10.1145/1941553.1941564

Min Feng, Rajiv Gupta, and Iulian Neamtiu. 2012a. Effective parallelization of loops in the presence of I/O operations. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'12)*. ACM, New York, NY, 487–498. DOI:http://dx.doi.org/10.1145/2254064.2254122

Min Feng, Changhui Lin, and Rajiv Gupta. 2012b. PLDS: Partitioning linked data structures for parallelism. *ACM Trans. Archit. Code Optim.* 8, 4, Article 38 (Jan. 2012), 21 pages. DOI:http://dx.doi.org/10.1145/2086696.2086717

Manoj Franklin. 1993. *The Multiscalar Architecture*. Ph.D. Dissertation. University of Wisconsin at Madison, Madison, WI, USA. UMI Order No. GAX94-07362.

Manoj Franklin and Gurindar S. Sohi. 1996. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Trans. Comput.* 45, 5 (May 1996), 552–571. DOI:http://dx.doi.org/10.1109/12.509907

Stanley L. C. Fung and J. Gregory Steffan. 2006. Improving cache locality for thread-level speculation. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS'06)*. IEEE Computer Society, Washington, DC, 32–41. DOI:http://dx.doi.org/10.1109/IPDPS.2006.1639271

Lin Gao, Lian Li, Jingling Xue, and Pen-Chung Yew. 2013. SEED: A statically-greedy and dynamically-adaptive approach for speculative loop execution. *IEEE Trans. Comput.* 62, 5 (2013), 1004–1016. DOI:http://dx.doi.org/10.1109/TC.2012.41

Álvaro García-Yágüez, Diego R. Llanos, and Arturo González-Escribano. 2011. Exclusive squashing for thread-level speculation. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC'11)*. ACM, New York, NY, 275–276. DOI:http://dx.doi.org/10.1145/1996130.1996172

Alvaro Garcia-Yaguez, Diego R. Llanos, and Arturo Gonzalez-Escribano. 2014. Squashing alternatives for software-based speculative parallelization. *IEEE Trans. Comput.* 63, 7 (2014), 1826–1839.

María Jesús Garzarán, Milos Prvulovic, José María Llabería, Víctor Viñals, Lawrence Rauchwerger, and Josep Torrellas. 2005. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM Trans. Archit. Code Optim.* 2, 3 (Sept. 2005), 247–279. DOI:http://dx.doi.org/10.1145/1089008.1089010

María Jesús Garzarán, Milos Prvulovic, Víctor Viñals, José María Llabería, Lawrence Rauchwerger, and Josep Torrellas. 2003. Using software logging to support multi-version buffering in thread-level speculation. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*. IEEE Computer Society, Washington, DC, 170. http://dl.acm.org/citation.cfm?id=942806.943847

S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. 1998. Speculative versioning cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA'98)*. IEEE Computer Society, Washington, DC, 195–205. DOI:http://dx.doi.org/10.1109/HPCA.1998.650559

Rui Guo, Hong An, Ruiling Dou, Ming Cong, Yaobin Wang, and Qi Li. 2008. LogSPoTM: A scalable thread level speculation model based on transactional memory. In *Computer Systems Architecture Conference, 2008. ACSAC 2008. 13th Asia-Pacific*. IEEE Computer Society, Washington, DC, 1–8. DOI:http://dx.doi.org/10.1109/APCSAC.2008.4625443

Manish Gupta and Rahul Nim. 1998. Techniques for speculative run-time parallelization of loops. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC'98)*. IEEE Computer Society, Washington, DC, 1–12. http://dl.acm.org/citation.cfm?id=509058.509070

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE Computer Society, Washington, DC, 3–14. DOI:http://dx.doi.org/10.1109/WWC.2001.990739

Robert H. Halstead, Jr. 1985. MULTILISP: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.* 7, 4 (Oct. 1985), 501–538. DOI:http://dx.doi.org/10.1145/4472.4478

Per Hammarlund and et al. 2014. Haswell: The fourth-generation Intel core processor. *IEEE Micro* 2 (2014), 6–20.

Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. 2000. The Stanford Hydra CMP. *IEEE Micro* 20, 2 (March 2000), 71–84. DOI:http://dx.doi.org/10.1109/40.848474

Lance Hammond, Mark Willey, and Kunle Olukotun. 1998. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*. ACM, New York, NY, 58–69. DOI:http://dx.doi.org/10.1145/291069.291020

John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. DOI:http://dx.doi.org/10.1145/1186736.1186737

John L. Henning. 2007. SPEC CPU suite growth: An historical perspective. *SIGARCH Comput. Archit. News* 35, 1 (March 2007), 65–68. DOI:http://dx.doi.org/10.1145/1241601.1241615

Maurice Herlihy, J. Eliot, and B. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*. ACM, New York, NY, 289–300. DOI:http://dx.doi.org/10.1145/165123.165164

Ben Hertzberg and Kunle Olukotun. 2009. DBT86: A dynamic binary translation research framework for the CMP era. In *Proceedings of the 2nd Workshop on Parallel Execution of Sequential Programs on Multi-Core Architectures (PESPMA'09)*. 41–46.

Ben Hertzberg and Kunle Olukotun. 2011. Runtime automatic speculative parallelization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'11)*. IEEE Computer Society, Washington, DC, 64–73. http://dl.acm.org/citation.cfm?id=2190025.2190054

Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. 1992. Factoring: A method for scheduling parallel loops. *Commun. ACM* 35, 8 (1992), 90–101.

Nikolas Ioannou and Marcelo Cintra. 2011. Complementing user-level coarse-grain parallelism with implicit speculative parallelism. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, New York, NY, 284–295.

N. Ioannou, J. Singer, S. Khan, P. Xekalakis, P. Yiapanis, A. Pocock, G. Brown, M. Luján, I. Watson, and M. Cintra. 2010. Toward a more accurate understanding of the limits of the TLS execution paradigm. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*. IEEE Computer Society, Washington, DC, 1–12. DOI:http://dx.doi.org/10.1109/IISWC.2010.5649169

M. M. Islam, A. Busck, M. Engbom, S. Lee, Michel Dubois, and P. Stenstrom. 2007a. Limits on thread-level speculative parallelism in embedded applications. In *Eleventh Workshop on Interaction between Compilers and Computer Architectures (INTERACT-11)*. 10.

M. M. Islam, A. Busck, M. Engbom, S. Lee, Michel Dubois, and P. Stenstrom. 2007b. Loop-level speculative parallelism in embedded applications. In *Parallel Processing, 2007. ICPP 2007. International Conference on*. IEEE Computer Society, Washington, DC, 3–13. DOI:http://dx.doi.org/10.1109/ICPP.2007.53

Quinn Jacobson, Steve Bennett, Nikhil Sharma, and James E. Smith. 1997a. Control flow speculation in multiscalar processors. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA'97)*. IEEE Computer Society, Washington, DC, 218–. http://dl.acm.org/citation.cfm?id=548716.822688

Quinn Jacobson, Eric Rotenberg, and James E. Smith. 1997b. Path-based next trace prediction. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 30)*. IEEE Computer Society, Washington, DC, 14–23. http://dl.acm.org/citation.cfm?id=266800.266802

Quinn Jacobson and James E. Smith. 2000. Trace preconstruction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*. ACM, New York, NY, 37–46. DOI:http://dx.doi.org/10.1145/339647.339653

Hakbeom Jang, Channoh Kim, and Jae W. Lee. 2013. Practical speculative parallelization of variable-length decompression algorithms. In *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'13)*. ACM, New York, NY, 55–64. DOI:http://dx.doi.org/10.1145/2465554.2465557

Alexandra Jimborean. 2012. *Adapting the Polytope Model for Dynamic and Speculative Parallelization*. Ph.D. Dissertation. University of Strasbourg, Strasbourg, France. Advisor(s) Clauss, Philippe.

Alexandra Jimborean, Philippe Clauss, Jean-Franois Dollinger, Vincent Loechner, and JuanManuel Martinez Caamaño. 2013. Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. *Int. J. Parallel Program.* 42, 4 (2013), 529–545. DOI:http://dx.doi.org/10.1007/s10766-013-0259-4

Alexandra Jimborean, Philippe Clauss, Benoît Pradelle, Luis Mastrangelo, and Vincent Loechner. 2012a. Adapting the polyhedral model as a framework for efficient speculative parallelization. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM, New York, NY, 295–296. DOI:http://dx.doi.org/10.1145/2145816.2145861

Alexandra Jimborean, Luis Mastrangelo, Vincent Loechner, and Philippe Clauss. 2012b. VMAD: An advanced dynamic program analysis and instrumentation framework. In *Proceedings of the 21st International Conference on Compiler Construction (CC'12)*. Springer-Verlag, Berlin, 220–239. DOI:http://dx.doi.org/10.1007/978-3-642-28652-0_12

Youngjoon Jo and Milind Kulkarni. 2010. Brief announcement: Locality-aware load balancing for speculatively-parallelized irregular applications. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*. ACM, New York, NY, 183–185. DOI:http://dx.doi.org/10.1145/1810479.1810516

Chuanle Ke, Lei Liu, Chao Zhang, Tongxin Bai, Brian Jacobs, and Chen Ding. 2011. Safe parallel programming using dynamic dependence hints. In *OOPSLA'11 Proceedings*. ACM, New York, NY, 243–258.

Arun Kejariwal, Milind Girkar, Xinmin Tian, Hideki Saito, Alexandru Nicolau, Alexander V. Veidenbaum, Utpal Banerjee, and Constantine D. Polychronopoulos. 2010a. Exploitation of nested thread-level speculative parallelism on multi-core systems. In *Proceedings of the 7th ACM International Conference on Computing Frontiers (CF'10)*. ACM, New York, NY, 99–100. DOI:http://dx.doi.org/10.1145/1787275.1787302

Arun Kejariwal, Milind Girkar, Xinmin Tian, Hideki Saito, Alexandru Nicolau, Alexander V. Veidenbaum, Utpal Banerjee, and Constantine D. Ppoluchronopoulos. 2010b. On the efficacy of call graph-level thread-level speculation. In *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW'10)*. ACM, New York, NY, 247–248. DOI:http://dx.doi.org/10.1145/1712605.1712645

Arun Kejariwal, Xinmin Tian, Milind Girkar, Wei Li, Sergey Kozhukhov, Utpal Banerjee, Alexander Nicolau, Alexander V. Veidenbaum, and Constantine D. Polychronopoulos. 2007. Tight analysis of the performance potential of thread speculation using spec CPU 2006. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*. ACM, New York, NY, 215–225. DOI:http://dx.doi.org/10.1145/1229428.1229475

Arun Kejariwal, Xinmin Tian, Wei Li, Milind Girkar, Sergey Kozhukhov, Hideki Saito, Utpal Banerjee, Alexandru Nicolau, Alexander V. Veidenbaum, and Constantine D. Polychronopoulos. 2006. On the performance potential of different types of speculative thread-level parallelism: The DL version of this article includes corrections that were not made available in the printed proceedings. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS'06)*. ACM, New York, NY, 24–. DOI:http://dx.doi.org/10.1145/1183401.1183407

Kirk Kelsey, Tongxin Bai, Chen Ding, and Chengliang Zhang. 2009. Fast track: A software system for speculative program optimization. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'09)*. IEEE Computer Society, Washington, DC, 157–168. DOI:http://dx.doi.org/10.1109/CGO.2009.18

Hanjun Kim, Nick P. Johnson, Jae W. Lee, Scott A. Mahlke, and David I. August. 2012. Automatic speculative DOALL for clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, New York, NY, 94–103.

Hanjun Kim, Arun Raman, Feng Liu, Jae W. Lee, and David I. August. 2010. Scalable speculative parallelization on commodity clusters. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Los Alamitos, CA, 3–14.

Tom Knight. 1986. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming (LFP'86)*. ACM, New York, NY, 105–112. DOI:http://dx.doi.org/10.1145/319838.319854

Sai Charan Koduru, Min Feng, and Rajiv Gupta. 2013. Programming large dynamic data structures on a DSM cluster of multicores. In *7th International Conference on PGAS Programming Models*. 126.

Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. 2005. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro* 25, 2 (March 2005), 21–29. DOI:http://dx.doi.org/10.1109/MM.2005.35

Venkata Krishnan and Josep Torrellas. 1998. Executing sequential binaries on a clustered multithreaded architecture with speculation support. In *Proceedings of the 1998 Fourth International Symposium on High-Performance Computer Architecture (HPCA'98)*. IEEE Computer Society, Washington, DC, USA.

V. Krishnan and J. Torrellas. 1999. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Comput.* 48, 9 (1999), 866–880. DOI:http://dx.doi.org/10.1109/12.795218

V. P. Krothapalli and P. Sadayappan. 1988. An approach to synchronization for parallel computing. In *Proceedings of the 2nd International Conference on Supercomputing (ICS'88)*. ACM, New York, NY, 573–581. DOI:http://dx.doi.org/10.1145/55364.55420

V. P. Krothapalli and P. Sadayappan. 1990. Dynamic scheduling of DOACROSS loops for multiprocessors. In *Databases, Parallel Architectures and Their Applications,. PARBASE-90, International Conference on*. IEEE Computer Society, Washington, DC, 66–75. DOI:http://dx.doi.org/10.1109/PARBSE.1990.77118

C. P. Kruskal and A. Weiss. 1985. Allocating independent subtasks on parallel processors. *IEEE Trans. Softw. Eng.* SE-11, 10 (1985), 1001–1016.

M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. 2009a. Lonestar: A suite of parallel irregular programs. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE Computer Society, Washington, DC, 65–76. DOI:http://dx.doi.org/10.1109/ISPASS.2009.4919639

Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. 2009b. How much parallelism is there in irregular applications? In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)*. ACM, New York, NY, 3–14. DOI:http://dx.doi.org/10.1145/1504176.1504181

Milind Kulkarni, Patrick Carribault, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. 2008. Scheduling strategies for optimistic parallel execution of irregular programs. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures (SPAA'08)*. ACM, New York, NY, 217–228. DOI:http://dx.doi.org/10.1145/1378533.1378575

Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, and Keshav Pingali. 2011. Exploiting the commutativity lattice. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, New York, NY, 542–555. DOI:http://dx.doi.org/10.1145/1993498.1993562

Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. 2008. Optimistic parallelism benefits from data partitioning. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, 233–243. DOI:http://dx.doi.org/10.1145/1346281.1346311

Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic parallelism requires abstractions. In *PLDI 2007 Proceedings*. ACM, New York, NY, 211–222.

Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2009. Optimistic parallelism requires abstractions. *Commun. ACM* 52, 9 (Sept. 2009), 89–97. DOI:http://dx.doi.org/10.1145/1562164.1562188

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO'04)*. IEEE Computer Society, Washington, DC, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

Peng Li and Song Guo. 2010. Energy minimization on thread-level speculation in multicore systems. In *Proceedings of the 2010 Ninth International Symposium on Parallel and Distributed Computing (ISPDC'10)*. IEEE Computer Society, Washington, DC, 125–132. DOI:http://dx.doi.org/10.1109/ISPDC.2010.17

Xiao-Feng Li, ZhaoHui Du, Chen Yang, Chu-Cheow Lim, and Tin-Fook Ngai. 2005. Speculative parallel threading architecture and compilation. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops (ICPPW'05)*. IEEE Computer Society, Washington, DC, 285–294. DOI:http://dx.doi.org/10.1109/ICPPW.2005.81

Xiao-Feng Li, Zhao-Hui Du, Qingyu Zhao, and Tin-Fook Ngai. 2003. Software value prediction for speculative parallel threaded computations. In *First valie Prediction Workshop*. 18–25.

Shaoshan Liu, Christine Eisenbeis, and Jean-Luc Gaudiot. 2010. Speculative execution on GPU: An exploratory study. In *Proceedings of the 2010 39th International Conference on Parallel Processing (ICPP'10)*. IEEE Computer Society, Washington, DC, 453–461. DOI:http://dx.doi.org/10.1109/ICPP.2010.53

Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. 2006. POSH: A TLS compiler that exploits program structure. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*. ACM, New York, NY, 158–167. DOI:http://dx.doi.org/10.1145/1122971.1122997

Diego R. Llanos, David Orden, and Belén Palop. 2007. New scheduling strategies for randomized incremental algorithms in the context of speculative parallelization. *IEEE Trans. Comput.* 56, 6 (2007), 839–852. DOI:http://dx.doi.org/10.1109/TC.2007.1030

Diego R. Llanos, David Orden, and Belén Palop. 2008. Just-in-time scheduling for loop-based speculative parallelization. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on* 0 (2008), 334–342. DOI:http://dx.doi.org/10.1109/PDP.2008.13

Yangchun Luo, Wei-Chung Hsu, and Antonia Zhai. 2013. The design and implementation of heterogeneous multicore systems for energy-efficient speculative thread execution. *ACM Trans. Archit. Code Optim.* 10, 4, Article 26 (Dec. 2013), 29 pages. DOI:http://dx.doi.org/10.1145/2541228.2541233

Pedro Marcuello, Antonio Gonzalez, and Jordi Tubella. 1998. Speculative multithreaded processors. In *Proceedings of the 12th International Conference on Supercomputing (ICS'98)*. ACM, New York, NY, 77–84. DOI:http://dx.doi.org/10.1145/277830.277850

José F. Martínez and Josep Torrellas. 2002. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. ACM, New York, NY, 18–29. DOI:http://dx.doi.org/10.1145/605397.605400

Jan Martinsen, Hakan Grahn, and Anders Isberg. 2013. Using speculation to enhance javascript performance in web applications. *IEEE Internet Comput.* 17, 2 (March 2013), 10–19. DOI:http://dx.doi.org/10.1109/MIC.2012.146

Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. 2009. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. ACM, New York, NY, 166–176. DOI:http://dx.doi.org/10.1145/1542476.1542495

Mario Méndez-Lojo, Donald Nguyen, Dimitrios Prountzos, Xin Sui, M. Amber Hassaan, Milind Kulkarni, Martin Burtscher, and Keshav Pingali. 2010. Structure-driven optimizations for amorphous data-parallel programs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*. ACM, New York, NY, 3–14. DOI:http://dx.doi.org/10.1145/1693453.1693457

Samuel P. Midkiff and David A. Padua. 1987. Compiler algorithms for synchronization. *IEEE Trans. Comput.* 100, 12 (1987), 1485–1495.

R. Mirchandaney and J. H. Saltz. 1988. *Dodynamic: A Construct for On-the-Fly Parallelization of Loops*. Technical Report 650. Yale University. in preparation.

Cosmin E. Oancea, Alan Mycroft, and Tim Harris. 2009. A lightweight in-place implementation for software thread-level speculation. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures (SPAA'09)*. ACM, New York, NY, 223–232. DOI:http://dx.doi.org/10.1145/1583991.1584050

Kunle Olukotun, Lance Hammond, and Mark Willey. 1999. Improving the performance of speculatively parallel applications on the Hydra CMP. In *Proceedings of the 13th International Conference on Supercomputing (ICS'99)*. ACM, New York, NY, 21–30. DOI:http://dx.doi.org/10.1145/305138.305155

Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. 1996. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*. ACM, New York, NY, 2–11. DOI:http://dx.doi.org/10.1145/237090.237140

K. Olukotun, O. A. Olukotun, L. S. Hammond, and J. P. Laudon. 2007. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. Morgan & Claypool Publishers, San Rafael, CA. http://books.google.es/books?id=spZZDwuAgUYC

Jeffrey Oplinger, David Heine, Shih Liao, Basem A. Nayfeh, Monica S. Lam, and Kunle Olukotun. 1997. *Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor*. Technical Report. Stanford University, Stanford, CA.

Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. 1999. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*. IEEE Computer Society, Washington, DC, 303. http://dl.acm.org/citation.cfm?id=520793.825732

Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. 2005. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 10 Los Alamitos, CA, 5–118.

V. Packirisamy, Yangchun Luo, Wei-Lung Hung, A. Zhai, Pen-Chung Yew, and Tin-Fook Ngai. 2008. Efficiency of thread-level speculation in SMT and CMP architectures - performance, power and thermal perspective. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*. IEEE Computer Society, Washington, DC, 286–293. DOI:http://dx.doi.org/10.1109/ICCD.2008.4751875

Venkatesan Packirisamy, Shengyue Wang, Antonia Zhai, Wei-Chung Hsu, and Pen-Chung Yew. 2006. Supporting speculative multithreading on simultaneous multithreaded processors. In *Proceedings of the 13th International Conference on High Performance Computing (HiPC'06)*. Springer-Verlag, Berlin,, 148–158. DOI:http://dx.doi.org/10.1007/11945918_19

V. Packirisamy, A. Zhai, Wei-Chung Hsu, Pen-Chung Yew, and Tin-Fook Ngai. 2009. Exploring speculative parallelism in SPEC2006. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE Computer Society, Washington, DC, 77–88. DOI:http://dx.doi.org/10.1109/ISPASS.2009.4919640

Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. 2013. Trace based phase prediction for tightly-coupled heterogeneous cores. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, 445–456. DOI:http://dx.doi.org/10.1145/2540708.2540746

Roberto Palmieri, Francesco Quaglia, and Paolo Romano. 2011. Osare: Opportunistic speculation in actively replicated transactional systems. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*. IEEE, Los Alamitos, CA, 59–64.

Il Park, Babak Falsafi, and T. N. Vijaykumar. 2003. Implicitly-multithreaded processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*. ACM, New York, NY, 39–51. DOI:http://dx.doi.org/10.1145/859618.859624

Sanjay Jeram Patel, Marius Evers, and Yale N. Patt. 1998. Improving trace cache effectiveness with branch promotion and trace packing. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA'98)*. IEEE Computer Society, Washington, DC, 262–271. DOI:http://dx.doi.org/10.1145/279358.279394

A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. 2005. Measuring program similarity: Experiments with SPEC CPU benchmark suites. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005 (ISPASS'05)*. IEEE Computer Society, Washington, DC, 10–20. DOI:http://dx.doi.org/10.1109/ISPASS.2005.1430555

Christopher J. F. Pickett. 2007. Software speculative multithreading for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion (OOPSLA'07)*. ACM, New York, NY, 929–930. DOI:http://dx.doi.org/10.1145/1297846.1297950

Christopher J. F. Pickett and Clark Verbrugge. 2005. SableSpMT: A software framework for analysing speculative multithreading in java. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'05)*. ACM, New York, NY, 59–66. DOI:http://dx.doi.org/10.1145/1108792.1108809

Christopher J. F. Pickett and Clark Verbrugge. 2006. Software thread level speculation for the java language and virtual machine environment. In *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing (LCPC'05)*. Springer-Verlag, Berlin, 304–318. DOI:http://dx.doi.org/10.1007/978-3-540-69330-7_21

C. D. Polychronopoulos and D. J. Kuck. 1987. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.* 36, 12 (Dec. 1987), 1425–1439. DOI:http://dx.doi.org/10.1109/TC.1987.5009495

Leo Porter, Bumyong Choi, and Dean M. Tullsen. 2009. Mapping out a path from hardware transactional memory to speculative multithreading. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. IEEE Computer Society, Washington, DC, 313–324. DOI:http://dx.doi.org/10.1109/PACT.2009.37

Manohar K. Prabhu and Kunle Olukotun. 2003. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*. ACM, New York, NY, 1–12. DOI:http://dx.doi.org/10.1145/781498.781500

Manohar K. Prabhu and Kunle Olukotun. 2005. Exposing speculative thread parallelism in SPEC2000. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*. ACM, New York, NY, 142–152. DOI:http://dx.doi.org/10.1145/1065944.1065964

Prakash Prabhu, Ganesan Ramalingam, and Kapil Vaswani. 2010. Safe programmable speculative parallelism. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, New York, NY, 50–61. DOI:http://dx.doi.org/10.1145/1806596.1806603

Dimitrios Prountzos, Roman Manevich, Keshav Pingali, and Kathryn S. McKinley. 2011. A shape analysis for optimizing parallel graph programs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. ACM, New York, NY, 159–172. DOI:http://dx.doi.org/10.1145/1926385.1926405

Joan Puiggali, Boleslaw K. Szymanski, Teo Jové, and Jose L. Marzo. 2012. Dynamic branch speculation in a speculative parallelization architecture for computer clusters. *Concurr. Comput.: Pract. Exper.* 25, 7 (2012).

Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. 2005. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, NY, 269–279. DOI:http://dx.doi.org/10.1145/1065010.1065043

Ravi Rajwar and James R. Goodman. 2001. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 34)*. IEEE Computer Society, Washington, DC, 294–305. http://dl.acm.org/citation.cfm?id=563998.564036

Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. 2010. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, 65–76. DOI:http://dx.doi.org/10.1145/1736020.1736030

Easwaran Raman, Neil Vahharajani, Ram Rangan, and David I. August. 2008. Spice: Speculative parallel iteration chunk execution. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'08)*. ACM, New York, NY, 175–184. DOI:http://dx.doi.org/10.1145/1356058.1356082

Lawrence Rauchwerger. 2011. Speculative parallelization of loops. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer, New York, NY, 1901–1912. DOI:http://dx.doi.org/10.1007/978-0-387-09766-4_35

Lawrence Rauchwerger and David Padua. 1995. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *SIGPLAN Not.* 30, 6 (June 1995), 218–232. DOI:http://dx.doi.org/10.1145/223428.207148

L. Rauchwerger and D. A. Padua. 1999. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. Parallel Distrib. Syst.* 10, 2 (1999), 160–180.

Jose Renau, Karin Strauss, Luis Ceze, Wei Liu, Smruti Sarangi, James Tuck, and Josep Torrellas. 2005. Thread-level speculation on a CMP can be energy efficient. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS'05)*. ACM, New York, NY, 219–228. DOI:http://dx.doi.org/10.1145/1088149.1088178

J. Renau, K. Strauss, L. Ceze, Wei Liu, S. R. Sarangi, J. Tuck, and J. Torrellas. 2006. Energy-efficient thread-level speculation. *IEEE Micro* 26, 1 (2006), 80–91. DOI:http://dx.doi.org/10.1109/MM.2006.11

Jose Renau, James Tuck, Wei Liu, Luis Ceze, Karin Strauss, and Josep Torrellas. 2005. Tasking with out-of-order spawn in TLS chip multiprocessors: Microarchitecture and compilation. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS'05)*. ACM, New York, NY, 179–188. DOI:http://dx.doi.org/10.1145/1088149.1088173

Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. 1995. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. Program. Lang. Syst.* 17, 2 (March 1995), 233–263. DOI:http://dx.doi.org/10.1145/201059.201065

Eric Rotenberg, Steve Bennett, and James E. Smith. 1996. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 29)*. IEEE Computer Society, Washington, DC, 24–35. http://dl.acm.org/citation.cfm?id=243846.243854

Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. 1997. Trace processors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 30)*. IEEE Computer Society, Washington, DC, 138–148. http://dl.acm.org/citation.cfm?id=266800.266814

Eric Rotenberg and Jim Smith. 1999. Control independence in trace processors. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 32)*. IEEE Computer Society, Washington, DC, 4–15. http://dl.acm.org/citation.cfm?id=320080.320084

Amitabha Roy, Steven Hand, and Tim Harris. 2009. A runtime system for software lock elision. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys'09)*. ACM, New York, NY, 261–274. DOI:http://dx.doi.org/10.1145/1519065.1519094

Peter Rundberg and Per Stenström. 2000. Low-cost thread-level data dependence speculation on multiprocessors. In *Workshop on Scalable Shared Memory Multiprocessors*.

J. H. Saltz and R. Mirchandaney. 1988. *How to Schedule Complex Loops in Parallel*. Technical Report 657. Yale University.

Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. 1991. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.* 40, 5 (1991), 603–612.

Mehrzad Samadi, Amir Hormati, Janghaeng Lee, and Scott Mahlke. 2012. Paragon: Collaborative speculative loop execution on GPU and CPU. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units (GPGPU-5)*. ACM, New York, NY, 64–73. DOI:http://dx.doi.org/10.1145/2159430.2159438

Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. 1995. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*. ACM, New York, NY, 414–425. DOI:http://dx.doi.org/10.1145/223982.224451

J. Steffan and T. Mowry. 1998. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA'98)*. IEEE Computer Society, Washington, DC, 2. http://dl.acm.org/citation.cfm?id=822079.822712

J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. 2005. The STAMPede approach to thread-level speculation. *ACM Trans. Comput. Syst.* 23, 3 (Aug. 2005), 253–300. DOI:http://dx.doi.org/10.1145/1082469.1082471

J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. 2000. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*. ACM, New York, NY, 1–12. DOI:http://dx.doi.org/10.1145/339647.339650

J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. 2002. Improving value communication for thread-level speculation. In *Proceedings of the 8th International Symposium on*

High-Performance Computer Architecture (HPCA'02). IEEE Computer Society, Washington, DC, 65–. http://dl.acm.org/citation.cfm?id=874076.876480

Peiyi Tang and Pen-Chung Yew. 1986. Processor self-scheduling for multiple-nested parallel loops. In *ICPP*, Vol. 86. CRC Press, Boca Raton, FL, 528–535.

YuXing Tang, Kun Deng, and XingMing Zhou. 2005. The design space of CMP vs. SMT for high performance embedded processor. In *Embedded Software and Systems*, LaurenceT. Yang, Xingshe Zhou, Wei Zhao, Zhaohui Wu, Yian Zhu, and Man Lin (Eds.). Lecture Notes in Computer Science, Vol. 3820. Springer, Berlin, 30–38. DOI:http://dx.doi.org/10.1007/11599555_6

Chen Tian, Min Feng, and Rajiv Gupta. 2010a. Speculative parallelization using state separation and multiple value prediction. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM'10)*. ACM, New York, NY, 63–72. DOI:http://dx.doi.org/10.1145/1806651.1806663

Chen Tian, Min Feng, and Rajiv Gupta. 2010b. Supporting speculative parallelization in the presence of dynamic data structures. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, New York, NY, 12. DOI:http://dx.doi.org/10.1145/1806596.1806604

Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. 2008. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, Washington, DC, 330–341. DOI:http://dx.doi.org/10.1109/MICRO.2008.4771802

Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. 2009. Speculative parallelization of sequential loops on multicores. *Int. J. Parallel Program.* 37, 5 (Oct. 2009), 508–535. DOI:http://dx.doi.org/10.1007/s10766-009-0111-z

Chen Tian, Changhui Lin, Min Feng, and Rajiv Gupta. 2011. Enhanced speculative parallelization via incremental recovery. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. ACM, New York, NY, 189–200. DOI:http://dx.doi.org/10.1145/1941553.1941580

Josep Torrellas. 2011. Speculation, thread-level. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer, New York, NY, 1894–1900. DOI:http://dx.doi.org/10.1007/978-0-387-09766-4_170

Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F. P. O'Boyle. 2009. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In *ACM Sigplan Notices*, Vol. 44. ACM, New York, NY, 177–187.

Jordi Tubella and Antonio Gonzalez. 1998. Control speculation in multithreaded processors through dynamic loop detection. In *Proceedings of the 1998 Fourth International Symposium on High-Performance Computer Architecture (HPCA'98)*. IEEE Computer Society, Washington, DC, 14–23. DOI:http://dx.doi.org/10.1109/HPCA.1998.650542

Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA'96)*. ACM, New York, NY, 191–202. DOI:http://dx.doi.org/10.1145/232973.232993

Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. 1998. Simultaneous multithreading: Maximizing on-chip parallelism. In *25 Years of the International Symposia on Computer Architecture (Selected Papers) (ISCA'98)*. ACM, New York, NY, 533–544. DOI:http://dx.doi.org/10.1145/285930.286011

Theo Ungerer, Borut Robič, and Jurij Šilc. 2003. A survey of processors with explicit multithreading. *ACM Comput. Surv.* 35, 1 (March 2003), 29–63. DOI:http://dx.doi.org/10.1145/641865.641867

T. N. Vijaykumar. 1998. *Compiling for the Multiscalar Architecture*. Ph.D. Dissertation. The University of Wisconsin - Madison. AAI9813127.

T. N. Vijaykumar and Gurindar S. Sohi. 1998. Task selection for a multiscalar processor. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 31)*. IEEE Computer Society Press, Los Alamitos, CA, 81–92. http://dl.acm.org/citation.cfm?id=290940.290963

Steven Wallace, Brad Calder, and Dean M. Tullsen. 1998. Threaded multiple path execution. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA'98)*. IEEE Computer Society, Washington, DC, 238–249. DOI:http://dx.doi.org/10.1145/279358.279392

Shengyue Wang, Xiaoru Dai, Kiran S. Yellajyosula, Antonia Zhai, and Pen-Chung Yew. 2006. Loop selection for thread-level speculation. In *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing (LCPC'05)*. Springer-Verlag, Berlin, 289–303. DOI:http://dx.doi.org/10.1007/978-3-540-69330-7_20

Fredrik Warg and Per Stenström. 2001. Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*. IEEE Computer Society, Washington, DC, 221–230. http://dl.acm.org/citation.cfm?id=645988.674160

Fredrik Warg and Per Stenström. 2003. Improving speculative thread-level parallelism through module run-length prediction. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS'03)*. IEEE Computer Society, Washington, DC, 12.2. http://dl.acm.org/citation.cfm?id=838237.838521

Fredrik Warg and Per Stenström. 2005. Reducing misspeculation overhead for module-level speculative execution. In *Proceedings of the 2nd Conference on Computing Frontiers (CF'05)*. ACM, New York, NY, 289–298. DOI:http://dx.doi.org/10.1145/1062261.1062310

Michael E. Wolf and Monica S. Lam. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.* 2, 4 (1991), 452–471.

Peng Wu, Arun Kejariwal, and Călin Caşcaval. 2008. Compiler-driven dependence profiling to guide program parallelization. In *Languages and Compilers for Parallel Computing*, José Nelson Amaral (Ed.). Springer-Verlag, Berlin, 232–248. DOI:http://dx.doi.org/10.1007/978-3-540-89740-8_16

P. Xekalakis and M. Cintra. 2010. Handling branches in TLS systems with multi-path execution. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE Computer Society, Washington, DC, 1–12. DOI:http://dx.doi.org/10.1109/HPCA.2010.5416632

Polychronis Xekalakis, Nikolas Ioannou, and Marcelo Cintra. 2009. Combining thread level speculation, helper threads and runahead execution. In *Proceedings of the 23rd International Conference on Supercomputing (ICS'09)*. ACM, New York, NY, 410–420. DOI:http://dx.doi.org/10.1145/1542275.1542333

Polychronis Xekalakis, Nikolas Ioannou, and Marcelo Cintra. 2012. Mixed speculative multithreaded execution models. *ACM Trans. Archit. Code Optim.* 9, 3, Article 18 (oct 2012), 26 pages. DOI:http://dx.doi.org/10.1145/2355585.2355591

Polychronis Xekalakis, Nikolas Ioannou, Salman Khan, and Marcelo Cintra. 2010. Profitability-based power allocation for speculative multithreaded systems. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, Los Alamitos, CA, 1–11.

Paraskevas Yiapanis, Demian Rosas-Ham, Gavin Brown, and Mikel Luján. 2013. Optimizing software runtime systems for speculative parallelization. *ACM Trans. Archit. Code Optim.* 9, 4, Article 39 (Jan. 2013), 27 pages. DOI:http://dx.doi.org/10.1145/2400682.2400698

Chao Zhang, Chen Ding, Xiaoming Gu, Kirk Kelsey, Tongxin Bai, and Xiaobing Feng. 2010. Continuous speculative program parallelization in software. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*. ACM, New York, NY, 335–336. DOI:http://dx.doi.org/10.1145/1693453.1693501

Chenggang Zhang, Guodong Han, and Cho-Li Wang. 2013. GPU-TLS: An efficient runtime for speculative loop parallelization on GPUs. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE Computer Society, Washington, DC, 120–127. DOI:http://dx.doi.org/10.1109/CCGrid.2013.34

Zhijia Zhao and Xipeng Shen. 2015. On-the-fly principled speculation for FSM parallelization. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, 619–630.

Zhijia Zhao, Bo Wu, and Xipeng Shen. 2012. Speculative parallelization needs rigor: Probabilistic analysis for optimal speculation of finite-state machine applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, New York, NY, 433–434. DOI:http://dx.doi.org/10.1145/2370816.2370882

Zhijia Zhao, Bo Wu, and Xipeng Shen. 2014. Challenging the embarrassingly sequential: Parallelizing finite state machine-based computations through principled speculation. In *ACM SIGARCH Computer Architecture News*, Vol. 42. ACM, New York, NY, 543–558.

Chuan-Qi Zhu and Pen-Chung Yew. 1987. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Trans. Softw. Eng.* SE-13, 6 (June 1987), 726–739. DOI:http://dx.doi.org/10.1109/TSE.1987.233477

Craig Zilles and Gurindar Sohi. 2002. Master/slave speculative parallelization. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 35)*. IEEE Computer Society Press, Los Alamitos, CA, 85–96. http://dl.acm.org/citation.cfm?id=774861.774871

Received May 2015; revised February 2016; accepted April 2016