

# Set-Congruence Dynamic Analysis for Thread-Level Speculation (TLS)

Cosmin E. Oancea and Alan Mycroft

Computer Laboratory, Cambridge University, Cambridge, CB3 0FD, UK,  
Cosmin.Oancea@cl.cam.ac.uk and Alan.Mycroft@cl.cam.ac.uk

**Abstract.** The move to multi-core has increased interest in parallelizing sequential programs. Classical dependency-based techniques, although successful for some classes of programs, often fail due to the one-sided (conservative) approximation of program behavior. Thread-level speculation enables increased parallelism by allowing out-of-order execution: correct dependences are ensured by run-time monitoring and possible rollbacks. Two-sided approximations of program behavior are now acceptable if the rollback ratio is kept small. We describe dynamic analysis, based on representing dependencies as modular congruences, which can be employed on-line or off-line. One, *thread partitioning*, efficiently enables loop iterations to be allocated to threads (and calculates the maximum effective concurrency); the other, *fine-grain memory partitioning*, calculates a hash function that reduces space overhead and performance loss due to TLS-metadata-based and cache-based task interference.

## 1 Introduction

Thread-level speculation (TLS) is a parallelization technique that allows the compiler to partition the program into concurrent threads even in the presence of dependencies. While important work examines hardware TLS [2, 10, 22, 21], this paper examines higher-level analysis than the one naturally done by hardware.

Current software-TLS approaches [5, 6, 18] exhibit heavy-transactional support, and can yield good speed-up when (i) the static compiler disambiguates<sup>1</sup> enough accesses to amortize the speculation overhead, and (ii) the iteration granularity is high enough to amortize the transactional overhead related to starting a new iteration. Their design assumes little about patterns of accesses to memory, merely that these accesses generate relatively few dependencies.

Lightweight models [17, 16] facilitates a compositional perspective to software-TLS: a coarse variable-based memory partitioning is performed first and separate optimized (adaptive) TLS models are employed on each partition to exploit regular access-patterns. In the latter's presence these models can be very effective even when most of the instructions require speculative support. These lightweight models perform best when memory accesses with a partition are regular (think linear strided), but cannot be proved so statically because (i) of perhaps few singular points and (ii) of static analysis hindrances such as: complex control-flow

---

<sup>1</sup> It is provable that no transactional support is needed.

and data-structures (increased abstraction level), potential aliasing. Evidence of these hindrances is illustrated by proposals for C language extensions that provide a special scope that guarantees the absence of cross-iteration dependencies [13]. Lightweight models require a *hash function*<sup>2</sup> which aims to reduce speculative storage without generating inter-thread conflicts.

This paper proposes a framework for dynamic analysis to guide the introduction of lightweight TLS models. The main idea is to use profile runs to build patterns capturing dependent iterations. This leads to two orthogonal techniques: First, the iteration space is partitioned based on dependent statement-instance pairs to the goal of executing dependent iterations on the same thread. Second, the data space is partitioned into (nearly) disjoint access patterns from threads, to the result of efficient TLS models’ *hash-functions*. While TLS-related optimizations [11, 23, 2] have focused so far on tuning the original code to enhance speed-up, we investigate here the equally important direction of fine-tuning the TLS model according to code’s access patterns (see Section 4.3 for speed-up results).

Previous profiling solutions for TLS were mainly aimed at (i) identifying suitable code for TLS (few dependencies) and designing flexible thread-formation schemes that delay thread-spawning to minimize violations [1, 12], (ii) inferring linear predictors [19, 2] for scalars which are very likely to violate dependencies, and (iii) developing TLS cost-models to predict speed-up [7]. This paper’s main contribution is to introduce (at a high-level) an address-based, set-congruence model and algebra that, to our knowledge, is the first that attempts to:

- compute a iteration-to-thread partitioning that respects frequent dependencies<sup>3</sup> and addresses the iteration granularity need;
- identify *coarse-grained memory partitions*, i.e. an exhaustive set of address ranges; access patterns for these may vary, thus assigning TLS models per partition is most effective in general;
- identify regular, *fine-grained* access-patterns and uses them to construct the hash functions that allow lightweight TLS models to be effective. The latter is only succinctly presented in Section 4 due to space constraints.

In principle, we are interested in both conservative – all events are modeled, and two-sided approximation – enough events are modeled that the cost of speculation failure is kept within cost bounds. With the latter, our analysis is light enough to be applied both off-line and on-line (just-in-time), as desired: the analysis is run on the profiled information corresponding to a *small* iteration window  $W$ , and the algorithms are  $O(n \log n)$  in the number of profiled addresses. (Regularity can also be verified on a conveniently far away window.)

Comparing with static approaches [15, 14], besides the obvious more conservative (and hence imprecise) trait these exhibit, we note two interesting differences: First, static approaches need to investigate how various loop-index variables are combined to form an array index, hence requiring complex, relational analysis. Since dynamic analysis looks directly at the accessed address, our model is simpler (non-relational flavor) while covering the exploitable cases. Second, our dynamic analysis may be applied to richer containers (linked lists,

---

<sup>2</sup> We abuse notation here: it describes regular accesses instead of randomizing.

<sup>3</sup> Executes dependent iterations on the same thread – enabled by *in-place* TLS models.

trees) than (only) arrays as long as the memory has a regular structure; Chilimbi and Larus’s work [3, 4] improves cache behavior by re-organizing memory to a similar regular structure that also facilitates our dynamic analysis.

The rest of the paper is structured as follows: Section 2 provides the background, motivation, states the general problem and compares with several classical approaches. Section 3 presents how the profiling information is gathered and introduces (formally and in detail) the thread partitioning analysis. Due to space constraints, Section 4 only briefly sketches the memory-partitioning analysis and presents speed-up results. Section 5 concludes the paper.

## 2 Background, General Problem, Related Work

This paper uses modular arithmetic significantly. We write  $\mathbb{Z}_n$  to mean the integers (mod  $n$ ) ( $\mathbb{Z}_n \equiv \{\{0, n, 2n, \dots\}, \{1, n + 1, 2n + 1, \dots\}, \dots, \{n - 1, 2n - 1, \dots\}\}$ ). Elements of  $\mathbb{Z}_n$ , are referred to as *cosets*. This section briefly introduces software-TLS, provides the motivation and states the general problem for our two dynamic analysis techniques, and compares them with related static approaches.

### 2.1 Software TLS

We give here the essential TLS information required to understand this paper; [2, 24, 6, 18] provide a more comprehensive perspective. TLS exploits code regions that expose good amount of parallelism but for which static analysis fails to guarantee safety. Under TLS threads execute out of order, and use software/hardware structures, referred as *speculative storage*, to record the necessary information to track the inter-thread dependencies and to revert to a *safe* point and restart the computation upon the occurrence of a *dependency violation (rollback recovery)*.

The thread executing the lowest numbered iteration of all is referred to as the *master* thread since it encapsulates both the correct sequential state and control-flow; the others are *speculative* threads since they may consume “dirty” values and cause rollbacks. *Serial-commit* TLS models [5, 6, 17] isolate the speculative from the global state: each thread buffers its write-accesses, and commits them when it becomes master, hence WAR and WAW dependencies are implicitly satisfied. *In-place* models [16, 8, 20] modify directly the program state, while still enforcing the sequential semantics. Important differences with respect to *serial commit* models are that (i) all types of dependencies (RAW, WAR and WAW) may generate violations, but (ii) they are scalable – in number of processors that may contribute to speed-up, and (iii) allow a more flexible iteration-to-thread partitioning (threads may execute non-consecutive groups of iterations, see later).

Finally, empirical results suggest that a software-TLS application requires an iteration’s granularity to be in the range of thousands of instructions: (i) big enough to amortize the speculative overhead corresponding to starting a new iteration, (ii) but not too big – so that the speculative storage is kept within reasonable bounds. As discussed in the next section, when the original loop does not provide enough granularity, we re-shape the loop in a fashion that preserves iterations’ execution locality. In this sense, we denote by  $W_{min}$  and  $W_{max}$  the minimal, maximal bounds for the number of consecutive, original iterations that are allowed to execute concurrently. A collateral, but important advantage of increasing iteration granularity is that it improves load-balance among threads.

## 2.2 Thread Partitioning – High Level View

Given a block  $B$  forming the body of the loop `for(int i=0; i<N; i++) B(i);` we would like to schedule the iterations  $B(i)$  for a multi-core processor. We denote by  $P$  the number of processors and by  $C$  the number of threads used to parallelize the program. (In general, maximal speed-up occurs when  $C \geq P$ ).

We assume we have profiled a window of  $W_{max}$  consecutive iterations. The general problem addressed in Section 3 is to find a repetitive structure ( $\pi$ ) that defines how iterations are assigned to threads so likely dependencies are satisfied.  $\pi : \{0, \dots, W-1\} \rightarrow \{0, \dots, C-1\}$  gives the mapping from  $W$  consecutive iterations to concurrent threads. Writing as usual  $\pi^{-1}(c) = \{i \mid \pi(i) = c\}$ , the iterations executed by thread  $j$  are simply  $\pi^{-1}(j)$ . Note that  $C$  and  $W$  are also analysis outputs; convenient values should maximize application’s available degree of parallelism, keep threads well (load) balanced, and provide TLS’s desired granularity ( $W_{min} \leq W \leq W_{max}$ ). With  $\pi$ ,  $W$  and  $C$  the loop is re-written as:

```
parfor(t=0; t<C; t++)
  for(k=0; k<N/W; k++) {
    for_each(j ∈ π-1(t)) B(k*W+j);
    cond_wait; /* required by TLS */ }
```

TLS application requires a loose synchronization between threads to keep the concurrent execution well-localized This is depicted via `cond_wait` that preserves the invariant that always, at most  $C$  consecutive “expanded” iterations execute concurrently ( $|k_i - k_j| < C$ ,  $0 \leq i, j < C$ , where  $k_i$  is  $k$ ’s value on thread  $i$ ).

We give two examples to illustrate forms of  $B$ , in which we assume  $P = 8$ . *The first example* takes  $B(i)$  to be `a[i+4] = a[i] + 2`, code that features cross-iteration dependencies of distance 4. Without considering the iteration-granularity factor, a possible result is  $C = 4$ ,  $W = 4$  and  $\pi^{-1}(j) = \{j\}$ , meaning that iterations  $j + 4\mathbb{Z} \equiv \{j, j + 4, j + 8, \dots\}$  execute on thread  $j$ . Note that, with this code, hardware-parallelism is only partially exploited: we use 4 threads on 8 processors. To increase the iteration granularity, we may choose  $W$  a convenient multiple of 4, say  $W = 16$  and have  $\pi^{-1}(j) = \{j, j + 4, j + 8, j + 12\}$ . (The first “expanded” iteration for thread 0 consists of the original iterations  $\{0, 4, 8, 12\}$ .) Note however that this way of increasing iteration-granularity is only applicable to *in-place* TLS models, since the write-back phase of a serial-commit model cannot be implemented in any effective way ( $W = 4$  for the serial-commit model).

*The second example* takes  $B(i)$  to be `a[i] = a[i] + 2`, and hence corresponds to (cross-iteration) dependency-free code. A possible result is  $C = 8$ ,  $W = 8$  and  $\pi^{-1}(j) = \{j\}$  (iterations  $j + 8\mathbb{Z}$  execute on thread  $j$ ). Increasing iteration granularity by a factor of 4, yields  $W = 32$  and  $\pi^{-1}(j) = \{4j, 4j+1, 4j+2, 4j+3\}$ , meaning that thread 0 executes iterations  $\{0, 1, 2, 3, 32, 33, 34, 35, \dots\}$  and so on. (The first “expanded” iteration for thread 0 consists of the original iterations  $\{0, 1, 2, 3\}$ .) This method of increasing iteration granularity is applicable to both *serial commit* and *in-place* TLS models. (The serial-commit phase operates as expected since the new iteration is formed from consecutive original iterations.)

## 2.3 Exploiting Access-Patterns via Adaptive TLS Models

Oancea and Mycroft [17] argue that rather than applying one over-arching TLS model to parallelize an application, software flexibility is, in some cases, better

exploited by combining several lightweight TLS models [16,17], each protecting disjoint memory partitions. In principle, lightweight TLS models attempt to exploit a program’s access patterns and, where these exist, yield a very small memory overhead and hence good performance.

For illustration, we intuitively present a simple TLS technique to track RAW dependencies. Assume  $\text{LdVct}[]$  is a vector with as many entries as the size of an array  $\text{arr}[]$  that requires speculative support. A read from  $\text{arr}[i]$  in iteration  $r$  sets  $\text{LdVct}[i]=r$  iff  $r$  is currently the maximal iteration that has read  $\text{arr}[i]$ . A write to  $\text{arr}[i]$  by iteration  $w$  discovers a RAW violation when  $w < \text{LdVct}[i]$  since iteration  $\text{LdVct}[i]$  should have read the value written by  $w$ , but it did not.

To decrease speculative storage ( $\text{LdVct}$ ) size, a (not one-to-one) hash function, of form  $\text{hash}_{s,q,Q}(x) = ((x - s) \text{ quo } q) \text{ rem } Q$  can be used to map memory locations into indexes in  $\text{LdVct}$ . Now, the data space is partitioned into equivalence-classes ( $x_1 \sim x_2 \Leftrightarrow \text{hash}(x_1) = \text{hash}(x_2)$ ), and a speculative read/write operation is interpreted as if any locations belonging to the same equivalence class may have been read/written. Although the execution soundness is guaranteed for any such (not one-to-one)  $\text{hash}$ , good speed-up is achieved only when the number of false-positives ( $\text{hash}(x_1) = \text{hash}(x_2) \ \& \ x_1 \neq x_2$ ) leading to dependence violations is small, so that the additional rollback-recovery cost is vastly overcome by the small speculative memory-footprint and improved cache behavior. (Naively chosen  $\text{hashes}$  will likely translate to poor performance.)

Section 4 presents at a very high-level the analysis that determines  $\text{hash}$ ’s  $s, q$ , and  $Q$  parameters. Assuming a 32-bit word, the *first example* in Section 2.2, with  $B(i) \equiv \mathbf{a}[i + 4] = \mathbf{a}[i] + 2$ ,  $C = 4$ ,  $W = 16$  and  $\pi^{-1}(j) = \{j, j+4, j+8, j+12\}$ ,  $j \in \{0, \dots, 3\}$ , yields  $\text{hash}(x) = ((x - s) \text{ quo } 4) \text{ rem } 4$ , where  $s = a \text{ quo } 4$ , and  $a$  stands for the start address of array  $\mathbf{a}$ . One can verify that  $\text{hash}(x) \equiv i$ , for all addresses  $x$  accessed by thread  $i$ . Similarly, the *second example*, with  $B(i) \equiv \mathbf{a}[i] = \mathbf{a}[i] + 2$ ,  $C = 8$ ,  $W = 32$  and  $\pi^{-1}(j) = \{4j, 4j + 1, 4j + 2, 4j + 3\}$ ,  $j \in \{0, \dots, 7\}$ , yields  $\text{hash}(x) = ((x - s) \text{ quo } 16) \text{ rem } 8$ . One can verify that thread  $i$  accesses addresses that map to  $i$  via  $\text{hash}$ .

We can thus introduce speculation via a very small memory-overhead (the load/store vectors that track dependencies have sizes 4 and 8 for the two cases). Moreover, since a thread repetitively accesses the same index of  $\text{LdVct}$  (and different threads access different indexes) in the dependency tracking-structure we can obtain a cache-ideal layout of speculative storage.

## 2.4 Comparison with Static Analysis Techniques

The classical (static) treatment depends on the assumption that the loop-body  $B$  is simple in terms of (i) control flow – typically no conditionals, (ii) access-patterns – linear indexing, and (iii) used data-structures – basic type arrays, and (iv) provable no-aliasing. Where one of these does not hold, dependence analysis is likely to indicate sequential execution even on a multi-core processor. However, the dynamic behavior (in particular the data-dependencies) may in fact be reasonably regular, with a perhaps small number of exceptions; TLS allows the code parallelism to be extracted while providing the safety net with respect to these few exceptions. The dynamic analysis introduced in this paper optimizes TLS application: where strong regular behavior exists, lightweight, software-TLS models are effective even when most  $B$ ’s instructions require speculative support.

Our thread-partitioning analysis, presented in Sections 2.2 and 3, most closely resembles a form of octagonal analysis [15] but also using congruences [9]. Note also the difference that the traditional use of octagonal congruences is for analysing relationships between values of user variables while we analyse to determine values of the iteration number appearing at the ends of a run-time dependency ( $x - y = c \pmod{M}$  is a octogon-type congruence).

Our address-partitioning analysis, introduced in Section 2.3 and briefly presented in Section 4, is at a high level related to Masdupuy’s analysis of trapezoid congruences [14]. The latter is a complex framework for relational integer analysis, aimed at describing multi-dimensional array indexes, that leads to “interval-like” or “congruence-like” information when interval or congruence analysis is relevant, respectively. We employ a similar strategy aimed at reducing `hash`’s image cardinality (i.e.  $Q$ ), and thus TLS’s memory overhead, but we restrict our intervals to be equal-sized, since we need a fast `hash`. While the introduction has recounted several profiling-related approaches, other TLS-related optimizations include data-flow algorithms for identifying “idempotent references” [11], aggressive instruction scheduling techniques aiming at reducing the stalls associated with scalar values [23], and other optimizations related to loop inductors, light thread synchronization locks, and reduction operators [2].

### 3 Thread Partitioning

The analysis presented in this section (i) identifies the cross-loop dependencies that are likely to yield run-time violations, (ii) classifies dependencies into rare-events, which can be ignored, and (frequent) repetitive-events that need to be satisfied, and (iii) attempts to describe the iteration space via a *regular structure*, in which iterations involved in cross-loop dependencies are assigned to execute on the same thread, while maintaining the load-balance among threads. The latter is the most efficient method of satisfying frequent-dependencies.

Constructing the regular structure is more useful than merely representing the value set of addresses at a given program point because of the need to model dependencies. It therefore involves representing relationships between addresses occurring at two program points in different iterations – one the source of the dependency and one the target. Our analysis applies to both regular and irregular, and simple and nested loops. For simplicity, in the following we restrict our discussion to single loops and generalize in Section 3.6 for loop nests.

#### 3.1 Notations, Preliminaries and Profiling Instrumentation

For simplicity, throughout the paper, we discuss our profiling and analysis techniques in the context of loop parallelization, where threads concurrently execute iterations out of the program order. However, this can be easily generalized to any thread-partitioning, as long as partitions are numbered in a fashion that respects the total order imposed by the sequential program’s control flow.

We assume that a simple static-analysis is performed first, to identify the read/write accesses of memory locations that cannot be disambiguated and hence require speculative support. We refer to the latter as speculative program points (SPP). Note that a SPP is associated with either a write or a read access of

memory locations;  $\text{mode} : \text{SPP}_{dom} \rightarrow \{\mathbf{r}, \mathbf{w}\}$  represents this relation. We denote by  $\mathbf{A}_{dom}$ ,  $\mathbf{IS}_{dom}$  and  $\text{SPP}_{dom}$  the domains of valid addresses, loop iteration space, and SPP. Hence  $\mathbf{A}_{dom}$ ,  $\text{SPP}_{dom}$  and  $\mathbf{IS}_{dom} \subset \mathbb{Z}$ , where we consider iteration  $i$  to be the  $i^{\text{th}}$  executed iteration in sequential program order.

At run-time, we employ an instrumentation phase that, for each SPP, gathers address-iteration pairs (PIA) recording which addresses were read/written by which iterations. Hence  $\text{PIA}_{\mathbf{q}} \subset \{(a, i) \mid a \in \mathbf{A}_{dom} \text{ and } i \in \mathbf{IS}_{dom}\}$ ,  $\mathbf{q} \in \text{SPP}_{dom}$ .

The cross-iteration dependencies that may appear at run-time can be identified by analyzing the PIAs corresponding to SPP pairs (PPP). For example if  $(a, i_1) \in \text{PIA}_{\mathbf{q}_1}$ ,  $(a, i_2) \in \text{PIA}_{\mathbf{q}_2}$ ,  $i_1 < i_2$ ,  $\text{mode}(\mathbf{q}_1) = \mathbf{w}$ , and  $\text{mode}(\mathbf{q}_2) = \mathbf{r}$ , then we have a true-dependence (RAW) with the source being executed in iteration  $i_1$  and the sink in iteration  $i_2$ . This leads to the following definitions:

**Definition 1 (Dependency-Class Notation).** We denote by  $(it_{src}, it_{snk}, tp)$  the class of run-time, cross-iteration dependencies, such that the source/sink of the dependency (on some memory location) is executed by the iteration numbered  $it_{src}$  /  $it_{snk}$ , respectively, and  $tp \in \{t, a, o\}$  denotes the dependency type: true (RAW), anti (WAR), or output (WAW). By construction we have:  $it_{src} < it_{snk}$ .

**Definition 2 (ADDG).** The dependency classes introduced in Definition 1 induce a directed acyclic dependency graph (ADDG), in which nodes are iteration numbers, and edges are directed from dependency’s source to sink and are annotated with the type of the dependence  $(t, a, o)$ . Singleton nodes are eliminated (they correspond to no dependency or to iteration-independent dependencies).

After (just-in-time) profiling a number of iterations, we construct for each SPP pair  $((\mathbf{q}_1, \mathbf{q}_2) \in \text{PPP})$  their associated directed acyclic dependency graph ( $\text{ADDG}_{(\mathbf{q}_1, \mathbf{q}_2)}$ ), in which the singleton nodes are eliminated. Dependencies of distance greater than  $W_{max}$  are trimmed-out since they cannot result in run-time violations (see `cond_wait` in Section 2.2). This approach of constructing per-PPP ADDGs as opposed to one whole-loop ADDG is motivated by the intuition that the resulting ADDGs often correspond to simple access patterns that, in many cases, can be easily inferred and expressed through basic congruence formulas. To this end, we assume that *or conditionals* have been normalized via code cloning, as shown in Figure 1(b). Next section exemplifies our approach.

### 3.2 Example

The cross-iteration dependencies, for the code shown in Figure 1(a), fall in one of two categories: (i) dependencies that, if not synchronized, will result in frequent run-time violations, leading to poor performance, and (ii) dependencies that, at run-time, rarely violate the sequential program semantics.

With respect to the *first category*, we identify three dependency-patterns that may still allow parallelism to be effectively exploited. Typical examples are the dependencies (of distances D, 1 and 1) between PP1-PP2, PP1-PP3 and PP4-PP5, whose corresponding ADDGs are shown in Figure 2. The ADDG corresponding to PP4-PP4, representing output-dependencies of distance 128, is not shown separately, but is overlapped in Figure 2(c) (pattern similar to Figure 2(b).)

The *second category*, includes for example the dependencies corresponding to PPP PP1-PP6 and PP7-PP8. The former may cause a true-dependency of distance

<pre> const int D = 4; const int B = 128; 1 for(int i=D; i&lt;N; i++) { 2   a[i] = ... ;           // PP1 3   ...  = a[i-D];        // PP2 4   if (i%8 == 1) 5     ...  = a[i-1];      // PP3 6   a[i%B] = ... ;       // PP4 7   ...  = a[i%D];       // PP5 8   if(highlyUnlikelyCond()) 9     ...  = a[i-1];      // PP6 10  e[i] = ... ;          // PP7 11     e[N-i];            // PP8 } </pre>	<pre> if(cond1    cond2) { ... // PP1 } else { ... // PP2 } } BECOMES if(cond1) { ... // PP1 } else if(cond2) { ... // PP1' } else ... // PP2 </pre>
--	--

Fig. 1. (a) Motivating Example

(b) Branch Normalization

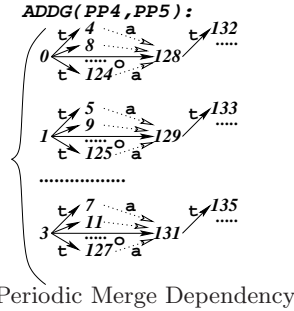
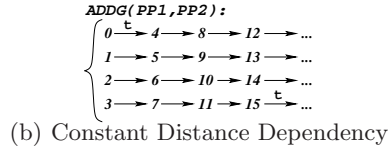
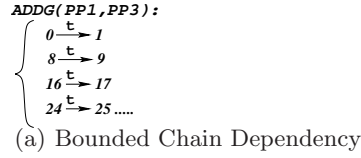


Fig. 2. ADDGs for Some Program Point (PP) Pairs in Figure 1.

1, but its sink is guarded by a condition that is highly-unlikely to evaluate to true. The latter PPP, if analyzed semantically, yields a group of cross-iteration, true/anti dependencies whose distances range uniformly from  $N$  to 0. Since we assume  $N \gg W_{max}$  they will cause very few run-time violations, roughly when the execution reaches the middle of the iteration space. (Note that a lightweight profiling approach may in fact not even discover these dependencies, which is consistent with our two-sided approximation strategy which ignores rare events.)

Ideally, we would like to fully exploit the available hardware parallelism, while introducing no explicit synchronization. For example, on a two-processor machine, an optimal thread-partitioning will execute iterations 0, 2, 4... on one thread, and iterations 1, 3, 5, ... on the other. This satisfies the observed dependencies without introducing any synchronization overhead, as the dependent instructions are executed on the same thread. On a four-processor machine it is probably better to use four threads, in which thread  $i$  executes iterations  $i, i + 4, i + 8, \dots$ , where  $0 \leq i < 4$ . This satisfies the dependencies in Figures 2(c) and 2(b), while light synchronization is introduced to satisfy the dependencies between threads 0 and 1 (those in Figures 2(a)).

### 3.3 Set-Congruence Model (in $\mathbb{Z} \times \mathbb{Z}$ )

As observed with the previous example, the ADDGs shown in Figure 2 have a repetitive structure that allows parallelism to be efficiently extracted (even under frequent dependencies). We aim at developing congruence relations such that:



- the repetitive structure is concisely and precisely described, and can be easily identified via pattern-matching type algorithms
- they can be effectively combined yielding a parallelization strategy that finds a good trade-off between the available code and hardware parallelism and the introduced synchronization.

**Definition 3 (Modulo/Step Operators).** Given  $0 \leq a, b < M$ , where  $a, b, M \in \mathbb{N}$ , we define the modulo operator  $\langle M \rangle$  and the step operator as:

$$(a, b)\langle M \rangle = \{(x, y) \mid x \equiv a \pmod{M} \text{ and } y \equiv b \pmod{M}\} \quad (\text{Figure 2(b)})$$

$$(a, b)|M \rangle = \begin{cases} \{(a + kM, b + kM) \mid k \in \mathbb{N}\}, & \text{if } a < b, \\ \{(a + kM, b + (k + 1)M) \mid k \in \mathbb{N}\}, & \text{if } a \geq b \end{cases} \quad (\text{Figure 2(a)})$$

$M$  is called characteristic. For  $M = 0$ ,  $(a, b)\langle 0 \rangle = (a, b)|0 \rangle = \{(a, b)\}$ . Finally, we lift the modulo/step operators (from pairs) to sets (of pairs):  $S\langle M \rangle = \cup_{(a,b) \in S} (a, b)\langle M \rangle$ , where  $S \in \mathbb{P}(\mathbb{Z}_M \times \mathbb{Z}_M)$ . The definition of  $S|M \rangle$  is similar.

Note that the step operator is more precise than the modulo operator:  $S|M \rangle \subseteq S\langle M \rangle$ . For example  $(0, 8) \in (0, 0)\langle 4 \rangle$  but  $(0, 8) \notin (0, 0)|4 \rangle$ . We represent the ADDG in Figure 2(a) as  $(0, 1)|8 \rangle$ , since there is no constraint that requires iterations 0 and 9 to be executed on the same thread, for example.

We represent the ADDG in Figure 2(c) via the modulo operator as  $\cup_{0 \leq i < 4} (i, i)\langle 4 \rangle$  (the step operator fails to represent it since, for example,  $(0, 8) \notin (0, 0)|4 \rangle$ ). Furthermore, the ADDG in Figure 2(b) also requires the modulo operator ( $\cup_{0 \leq i < 4} (i, i)|4 \rangle$ ) due to the implicit transitive closure: iterations 0 and 4, and 4 and 8 execute on the same thread, hence iterations 0 and 8 execute on the same thread (although iterations 0 and 8 are not dependent). (The transitive closure is a result of iterations 0, 4 and 8 belonging to the same ADDG's connected component.)

### 3.4 Set-Congruence Algebra (in $\mathbb{Z} \times \mathbb{Z}$ )

We present now how formulas describing ADDG's basic patterns are combined. The non-relational, static analysis of integer congruence properties employs the lattice of integer cosets to join same-variable formulas:  $(a_1 + b_1\mathbb{Z}) \sqcup (a_2 + b_2\mathbb{Z}) = a_1 + \gcd(b_1, b_2)\mathbb{Z}$  if  $\gcd(b_1, b_2) \mid (a_2 - a_1)$  and  $\mathbb{Z}$  otherwise. Applying our analysis on a special subset of  $\mathbb{Z} \times \mathbb{Z}$  (see Definition 3), results in a (different) formula to manipulate these descriptions.

**Definition 4 (Additive Subgroup).** We denote by  $[m]_M$  the additive subgroup of  $\mathbb{Z}_M$  generated by  $m$ . Note that  $[m]_M = [g]_M$ , where  $g = \gcd(m, M)$ , since both subgroups have the same cardinality  $M/g$  and  $g$  generates  $m$ .

Assume  $a < b$  and  $m_1 < m_2$ . The step relation yields the invariant:  
 $(a, b)|m_1 \rangle = \{(a + k*m_1, b + k*m_1) \mid k \in \mathbb{Z}\} \subseteq \{(a + e + k*m_2, b + e + k*m_2) \mid k \in \mathbb{Z}, e \in [m_1]_{m_2}\} = \cup_{e \in [m_1]_{m_2}} (a + e, b + e)|m_2 \rangle$ .

The modulo relation is similar to the integer congruence unification. Denoting  $m = \gcd(m_1, m_2)$  ( $[m_1]_{m_2} \equiv [m]_{m_2}$ ) leads to:  $(a, b)\langle m_1 \rangle \subseteq (a, b)\langle m \rangle$ . (Also  $(a, b)|m_1 \rangle \subseteq \cup_{e \in [m_1]_{m_2}} (a + e, b + e)|m_2 \rangle \subseteq (a, b)|m \rangle \subseteq (a, b)\langle m \rangle$ .)

We demonstrate now the usefulness of differentiating between step and modulo relations. Combining two congruence relations corresponds to taking the union of the sets they represent. For the modulo relation we have:  $\{(0, 1)\langle 8 \rangle \cup \{(0, 1)\langle 18 \rangle \subseteq \{(0, 1)\langle 2 \rangle$ , which implies that the program cannot be parallelized (as iterations 0 and 1 modulo 2 are executed on the same thread). However, with the step relation we get:  $\{(0, 1)\langle 8 \rangle \cup \{(0, 1)\langle 18 \rangle \subseteq S\langle 18 \rangle$ , where  $S = \{(0, 1), (8, 9), (16, 17), (6, 7), (14, 15), (4, 5), (12, 13), (2, 3), (10, 11)\}$ . In this case we can run 9 concurrent threads while satisfying the observed dependencies: thread  $i$  executes iterations  $S[i][0]$  and  $S[i][1] \bmod 18$ , where  $0 \leq i < 9$ . The next theorem formalizes these results.

**Theorem 1 (Step/Modulo Refining).** *Let  $U \subseteq \mathbb{P}(\mathbb{Z} \times \mathbb{Z})$  be of the form  $S\langle m \rangle$ , and  $g = \gcd(M, m)$ . The smallest set  $U', U \subseteq U'$ , of the form  $S'\langle M \rangle$  exists and is obtained when:  $S' = \{(x, y) \mid x \equiv a + e \text{ and } y \equiv b + \alpha * m + e \bmod M, \text{ where } (a, b) \in S, \alpha = 0 \text{ for } a < b \text{ and } 1 \text{ otherwise, and } e \in [g]_M\}$  For the modulo relation the set  $S'$  can also be computed, but  $S'\langle M \rangle \equiv S\langle g \rangle$ .*

*Proof.* Straightforward application of the finite additive subgroup theory.

The rest of this section gives the unification rules for the step/modulo operators: intuitively, they aim to obtain a formula of best precision (highest degree of parallelism) and conciseness (smallest characteristic), in this order.

**Definition 5 ( Degree of Parallelism ).** *Let  $l_{max}$  be the maximal number of nodes of a connected component of the ADDG induced by a formula of the form  $S\langle m \rangle$ , where only iterations  $0..(m-1)$  are considered. The degree of parallelism of  $S\langle m \rangle$  is  $\lceil m/l_{max} \rceil$ . The same holds for  $S\langle m \rangle$ .*

**Definition 6 ( Step/Modulo Unification ).**

*MS.1:*  $S_1\langle m \rangle \sqcup S_2\langle m \rangle = (S_1 \cup S_2)\langle m \rangle$

*MS.2:*  $S_1\langle m_1 \rangle \sqcup S_2\langle m_2 \rangle = ([S_1\langle m_1 \rangle]_m \cup [S_2\langle m_2 \rangle]_m)\langle m \rangle$ , where  $m \in \{m_2, \gcd(m_1, m_2)\}$  is the value that maximizes the degree of parallelism. In the case of equality, take the smaller  $m$ . Similarly, combining two step relations yields a step relation. Combining two modulo relations yields a modulo relation where the resulting characteristic is computed by taking the gcd.

### 3.5 Pattern Identification

Figure 2 gives three dependency patterns for a given PPP that allow speculative parallelism to be extracted. They hence constitute the basic building-blocks of our analysis; if no such pattern can be found then speculation is likely to be unhelpful. While it might appear natural to merely union the ADDGs for each PPP, in general this gives a complete ADDG from which is hard to extract such patterns. Instead, we determine potential patterns for each ADDG and unify ( $\sqcup$ ) their set-congruence formulas as described in the previous section.

For space reasons we do not give exact pattern-matching algorithms, instead we prefer to outline the process by (i) identifying the main pattern characteristics such an algorithm should identify, (ii) giving the pattern's congruence formula, and (iii) where not already discussed, present how formula unification is achieved.

**Bounded-Chain Dependency Pattern:** The first basic pattern corresponds to the one shown in Figure 2(a). An ADDG satisfies this pattern if (i) there are very many connected components, each containing a small number of nodes,  $l$  (typically 2), (ii) for any two dependencies in the same position on different connected components  $p_i \rightarrow q_i$  and  $p_j \rightarrow q_j$ ,  $m_{i,j} = p_i - q_i = p_j - q_j$ , and, (iii) denoting by  $m = \gcd\{m_{i,j}\}$ , the degree of parallelism  $\text{ParDeg} = \lceil m/l \rceil$  is big enough – if the latter is 1 for example, parallelism cannot be extracted, since we intend to execute the connected component’s nodes (iterations) on the same thread. The ADDG is finally described by  $R|m\rangle$ , where  $R \subseteq \mathbb{P}(\mathbb{Z}_m \times \mathbb{Z}_m)$  contains the iteration pairs corresponding to the edges of one connected component. (By pattern’s definition, all connected components generate the same  $S$ .) For example, the ADDG in Figure 2(a) formula is  $(0, 1)|8\rangle$ .

**Constant-Distance Dependency Pattern:** The second basic pattern corresponds to the one shown in Figure 2(b). An ADDG of  $l$  connected-components satisfies this pattern if letting  $m = \gcd\{j - i \mid i \rightarrow j \text{ is a dependency}\}$  then  $l < m$ , with  $m$  big enough. The ADDG is then described by  $R\langle m\rangle$ , where  $R$  is the root set of the ADDG (i.e. nodes that are the sink of no dependency).

**Periodic Merge Dependency Pattern:** This pattern is shown in Figure 2(c). An ADDG satisfies this pattern if there are several (few consecutive) nodes ( $[0, 3]\langle 128\rangle$  in Figure 2(c)) that are the source/sink of many dependencies, the structure is repetitive, and if eliminating these nodes from the graph results in only singleton nodes. Hence, if iterations  $[0, 3]\langle 128\rangle$  are executed sequentially (achieved via synchronization barriers), then iterations ( $[4, 127]\langle 128\rangle$ ) have no dependencies and can be executed in parallel, out of order. Due to space constraints we do not give the formula/unification rules for this pattern here. The intuition is that this pattern is orthogonal with the other two, hence we unify only between instances of this pattern: the semantics is that barriers are enforced while iterations in-between barriers are dispatched to threads as determined by the step/modulo unification formulas. A special case when simplification with a modulo-based pattern is possible and desirable is when the ADDG of this pattern is included in the one of the modulo-based pattern. (The ADDG in Figure 2(c)  $\subset \{(i, i) \mid 0 \leq i < 4\}\langle 4\rangle$ .) The unification trivially yields the modulo-based formula, thus eliminating the (unnecessary) barrier overhead.

### 3.6 Further Remarks

The time complexity of the analysis proposed in this section is dominated by the ADDGs construction phase, which is on average  $O(n \log n)$  in the number of profiled addresses<sup>4</sup>. Then, pattern-matching ADDGs to determine formulas is linear in the number of dependencies, while unifying formulas among ADDGs is cheap under the presented algebra. The order in which ADDG formulas are unified is important: a state-of-art framework should aim to assign iteration to threads such that most dependencies are resolved, while preserving the optimal degree of parallelism. Other PPPs whose unification would yield too conservative results are synchronized (see Section 3.2). These heuristics are not discussed here.

<sup>4</sup> Sort the per SPP addresses; constructing the ADDG for a PPP is then linear; we then construct only the ADDG corresponding to PPP whose address sets overlap.

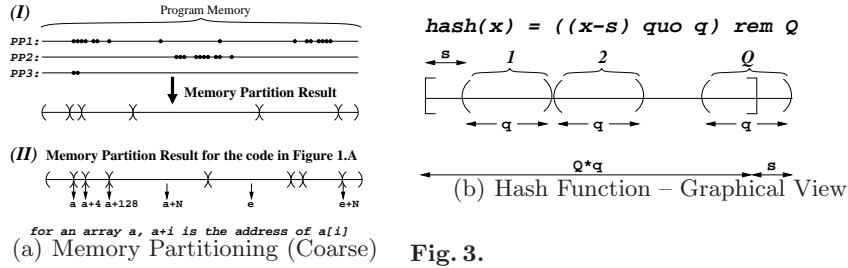


Fig. 3.

Finally, we have discussed our analysis so far in the context of simple-loops. To generalize to loop-nests, we represent iteration numbers in  $\mathbb{Z}^p$ , where  $p$  is the loop's nesting depth. We apply our analysis for the most-outermost loop,  $L$ , of suitable TLS granularity, by projecting  $\mathbb{Z}^p$  to  $\mathbb{Z}$ , in the context of  $L$ . If the analysis fails to give an acceptable result, we repeat it for inner loops.

## 4 Memory Partitioning

This section introduces at a high-level the second part of our analyzes, whose goals were stated in Section 2.3. Section 4.1 presents how memory is coarse-grained partitioned into disjoint intervals that intuitively correspond to different variables (e.g. arrays) whose accesses were not statically disambiguated. The most suitable type of TLS model is then chosen to protect each of these partitions (see [17] for the software composition techniques). The latter partitioning facilitates the fine-grained memory partitioning, presented in Section 4.2, by which each individual TLS model's hash function is computed based on coarse-partition's access-patterns. Finally, Section 4.3 presents speed-up results.

### 4.1 Building Variable-Based Memory Partitions

Figure 3(a).(I) intuitively depicts our approach. The profiling phase has yielded the set of addresses accessed at each program point. A clustering technique is then employed to exhaustively partition the memory into mostly-disjoint intervals, where a few exceptions are allowed (e.g. the two "singular" points of PP1).

For on-line analysis, when only a window of iterations is profiled, we compute a linear formula that predicts how intervals grow. The latter are used to compute interval boundaries. Note that (i) a SPP may correspond to two intervals (e.g. PP1), as it is not required that a variable is laid out contiguously in memory, and (ii) two SPPs may correspond to the same interval/partition (e.g. they may access the same variable). Also, it is preferable to split an interval  $I$  into  $I_1 \cup I_2$ , when we observe that one SPP repeatedly accesses only few addresses in  $I$  (e.g. PP3 forms  $I_1$ ) – this allows to more aggressively adapt/optimize  $I_2$ 's TLS model's behavior (small memory overhead within very few false-positives). For safe-languages (Java, C#) an (extra) orthogonal partitioning can be made based on type information or other invariants guaranteeing that two SPP refer to disjoint set of addresses (although their intervals overlap).

Figure 3(a).(II) shows the analysis result for the code in Figure 1. We denote by  $a$  and  $e$  the start address of arrays  $a$  and  $e$ . We observe that accesses of  $a$  form three disjoint partitions:  $[a, a+3]$  for PP5,  $[a+4, a+127]$  for PP4, and  $[a+128, ...]$

for PP1, PP2 and PP3. Accesses of  $e$  form three intervals: an increasing one starting at  $e$ , a decreasing one starting at  $e + N$  and a buffer-interval in the middle. While the first two partitions may use aggressive fine-grained partitioning, the middle-one should use a precise partitioning (i.e. hash function is one-to-one).

Having computed the coarse-grained memory partitions, we look at their associated ADDGs to determine the most suitable type of TLS model for them. We keep into account that serial commit models [17, 5, 6] implicitly satisfy WAW and WAR dependencies, while in-place models [16] do not. Also, [17] is inefficient in the presence of many iteration-independent RAW. However, note that if the thread partitioning requires a non-trivial characteristic ( $m \neq 0$ ), then only in-place models may be employed, and only then when a serial phase<sup>5</sup> is not needed.

## 4.2 Fine-Grained Memory Partitioning

For each memory partition we apply a congruence/set analysis that attempts to map addresses accessed by different threads into mostly disjoint (coset-based) equivalence classes. The latter naturally induces the TLS model hash function, and effectively reduces the TLS memory overhead while introducing very few false-positives. An important consequence of this strategy is that it implicitly optimizes the speculative-storage cache-behavior in that the same thread repeatedly accesses the same speculative storage elements (and different ones for different threads) and hence requires mostly  $L1$ -cache accesses (rather than  $L2$ ).

We are looking for hash functions of the form:  $\mathbf{hash}_{s,q,Q}(x) = ((x - s) \mathbf{quo} \ q) \mathbf{rem} \ Q$ , because (i) they are computationally effective (especially when  $q$  and  $Q$  are powers of 2) and (ii) they enable both an interval and congruence like analysis, which are both useful in reducing the image cardinal  $Q$ , and hence the speculative storage size. The hash function can be seen as an equivalence relation among addresses:  $a_1 \sim a_2 \Leftrightarrow \mathbf{hash}_{s,q,Q}(a_1) = \mathbf{hash}_{s,q,Q}(a_2)$ . Figure 3(b) graphically depicts this view: there are  $Q$  intervals of equal length  $q$ . All the addresses  $a$  such that  $(a - s) \mathbf{rem} \ (Q * q)$  belong to the  $i^{th}$  interval and are in one equivalence class. This class corresponds to the union of cosets:  $(i * q) + Q\mathbb{Z} \cup (i * q + 1) + Q\mathbb{Z} \cup \dots \cup (i * q + q - 1) + Q\mathbb{Z}$ . The use of the offset  $s$  is to align equivalence-classes in  $\mathbf{hash}_{0,1,Q*q}$  so that they can be safely collapsed by interval formation into  $\mathbf{hash}_{s,q,Q}$ .

Intuitively,  $\mathbf{hash}$  should satisfy the invariant that for any two SPPs that may generate a dependency violation (e.g. at least one is a write), the addresses accessed by iterations executed on different threads correspond mainly to different cosets of  $\mathbb{Z}_Q$ , where a few exceptions can be accommodated. In general this problem is computationally expensive to solve; although not presented here, we have developed a guided-search heuristic that, although does not guarantee an optimal solution, for all practical cases we encountered, it does so and is linear in the number of profiled addresses.

## 4.3 Speed-up Results

Table 1 shows speed-up results, computed as the ratio between sequential and parallel timings, for several applications selected from the BYTEmark and SciMark benchmarks (see also [16]). All the tests were performed on a SMP Sun

<sup>5</sup> This might still be required for scalar computation or IO operations.

Seq/Parallel	HandPar	OptROHash	OptHash	Naive	OneToOne
IDEA DeKey	3.83	2.78	2.44	0.96	0.65
IDEA Cipher	3.87	3.22	1.44	1.11	0.95
NeuralNetBW	1.64	1.15	1.07	0.90	0.25
NeuralNetFW	2.04	1.65	1.46	0.15	0.11
SparMatMult	2.11	1.93	1.60	0.57	0.13
FFT	2.02	1.90	1.90	0.66	0.66

**Table 1.** Speed-ups: Sequential / Parallel Timing Ratio(4 Processors)

machine with 8 Gb RAM memory, and four Opteron 850 processors, running Fedora Core 4. We used the gcc3.4.4 compiler at -O2 optimization level.

The *second column* represents the speed-up achieved for optimal, hand-based parallelization (no TLS overhead). The *third column* corresponds to applying TLS via the dynamic analysis presented in this paper. We have used two TLS models: SPLIP – an in-place model [16] and SPRO – the read-only model in which a read just *reads* the value (no time/space overhead), while a write causes a rollback followed by a sequential fix-up. SPLIP models use optimized **hashes**.

The **OptHash** column refers to the case when only SPLIP is employed, however on optimized **hashes**. This serves as base of comparison with the last two columns. The column **Naive** refers to a naively chosen **hash**, in which  $q=s=0$  and  $Q$  is roughly the size of the range of addresses accessed by concurrent iterations. The last column uses one-to-one **hashes**:  $q=s=0$  and  $Q$  is roughly the data-space size. (For **FFT** and **NeuralNetFW** columns 5 and 6 use roughly the same  $Q$ .)

The differences between optimized and (i) naively chosen and (ii) one-to-one **hashes** are significant for all tested applications. The reasons are: (i) the near-ideal cache behavior of optimized **hashes** (column 4 vs 5) and (ii) the small(er) memory overhead. The results for **IDEA DeKey** and **SparseMatMult** look somewhat surprising in that the differences between columns 3 and 4 are more pronounced than in the other cases. The reason is that the read-to-write ratio is very high (and SPRO is very effective). These differences also appear for columns 4 vs 5 because the naive version suffers from read-contention (concurrent cache eviction due to writes to TLS’s meta-data), while optimized **hashes** do not. A last observation is that when the application features bad cache-locality (last two benchmarks), but still a regular behavior, we can expect to obtain close to optimal (hand-parallelized) speed-up because the TLS overhead is furthermore amortized by the negative memory-hierarchy effects of the original program.

## 5 Conclusions

We have shown how dynamic analysis of addresses accessed during a loop can be used to facilitate thread-level speculation. First, we have presented an algebra for partitioning the iteration-space to threads such that repetitive dependencies are resolved and rare dependencies are ignored. Second, we have used dynamic analysis to fine-tune the TLS model to exploit code’s access patterns, as opposed to previous work, which has concentrated on optimizing the code for one TLS model. We achieved the latter by using coarse-grained followed by fine-grained partitioning of the data space; now *several optimized* TLS model instances are

employed to parallelize an application, instead of only *one, over-arching* model. Finally, we have presented results that validate the utility of our analysis.

## References

1. A. Bhowmik and M. Franklin. A General Compiler Framework for Speculative Multithreading. In *SPAA '02 Proceedings*. ACM, 2002.
2. M. K. Chen and K. Olukotun. The Jrpm System for Dynamically Parallelizing Java Programs. In *ISCA-30*, June 2003.
3. T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-Conscious Structure Definition. In *PLDI'99*.
4. T. M. Chilimbi and J. R. Larus. Using Generational Garbage Collection to Implement Cache-Conscious Data Placement. In *ISMM*, 1998.
5. M. Cintra and D. R. Llanos. Toward Efficient and Robust Software Speculative Parallelization on Multiprocessors. In *PPoPP'03*, June 2003, San Diego, California.
6. F. Dang, H. Yu, and L. Rauchwerger. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *IPDPS*, 2002.
7. J. Dou and M. Cintra. A Compiler Cost Model for Speculative Parallelization. In *ACM TACO*, vol. 4, no. 2, June 2007.
8. K. Fraser and T. Harris. Concurrent Programming Without Locks. *TOCS*, 2007.
9. P. Granger. Static Analysis of Linear Congruence Equalities among Variables of a Program. In *LNCS, Vol. 493*, 1991.
10. L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for Chip Multiprocessor. In *ASPLOS*, 1998.
11. S. Kim, C. Ooi, R. Eigenmann, B. Falsafi, and T. Vijaykumar. Reference Idempotency Analysis: A Framework for Optimizing Speculative Execution. In *PPoPP'01*.
12. W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: A TLS Compiler that Exploits Program Structure. In *PPoPP'06*.
13. A. Lokhmotov, A. Mycroft, and A. Richards. Delayed Side-Effects Ease Multi-Core Programming. In *Euro-Par'07, Rennes, France*.
14. F. Masdupuy. Array Operations Abstraction Using Semantic Analysis of Trapezoid Congruences. In *ICS '92*.
15. A. Miné. The Octogon Abstract Domain. In *Higher-Order and Symbolic Computation Journal, Vol. 19*, 2006.
16. C. E. Oancea and A. Mycroft. A Lightweight, In-Place Model for Software Thread-Level Speculation. In *email Cosmin.Oancea@cl.cam.ac.uk*.
17. C. E. Oancea and A. Mycroft. Software Thread-Level Speculation – An Optimistic Library Implementation. In *IWMSE*, 2008.
18. P. Rundberg and P. Stenstrom. An All-Software Thread-Level Data Dependence Speculation System for Multiprocs. *Journal of Instruction-Level Parallelism*, 1999.
19. Y. Sazeides and J. E. Smith. The Predictability of Data Values. In *MICRO 30: 30th International Symposium on Microarchitecture*, pages 248–258, 1997.
20. T. Shpeisman, V. Menon, A. Adl-Tabatabai, S. Balensiefer, S. Grossman, R. Hudson, K. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. In *PLDI'07*.
21. G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *ISCA-22*, pages 414–425, June 1995.
22. J. G. Steffan, C. G. Colohan, A. Zhai, and T. Mowry. A Scalable Approach for Thread Level Speculation. In *ISCA-27*, 2000.
23. A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In *ASPLOS X*, 2002.
24. C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Micro-35 Proceedings*. ACM, 2002.