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 can be successful for some classes of programs, but all too often such (static) analysis disallow parallelization because of the need for *safe* (one-sided) approximations of behaviour. Thread Level Speculation (TLS) enables increased parallelization by allowing out-of-order execution; correct dependencies are ensured by run-time monitoring and possible rollbacks. Two-sided approximations of program behaviour are now acceptable so long as the rollback ratio is kept small. We describe dynamic analysis based on representing dependencies as modular congruences. 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 which reduces performance loss due to TLS-metadata-based and cache-based task interference. The dynamic analysis can be used either on-line (during execution) or off-line (based on previous training runs).

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[1] 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 course 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

[1] It is provable that no transactional support is needed.

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 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*[2] 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. Section 4.3 shows speed-up results that demonstrate our analysis' utility.

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[3] 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.)

[2] We abuse notation here: it describes regular accesses instead of randomizing.
[3] Executes dependent iterations on the same thread – enabled by *in-place* TLS models.

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, 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, gives a non-trivial example, and presents speed-up results demonstrating our analysis usefulness. Section 5 concludes the paper.

2 Background, General Problem, Related Work

This paper uses modular arithmetic (over multiple modular bases) significantly. We write $\mathbb{Z}_n$ to mean the integers (mod n), or more formally $\mathbb{Z}_n \equiv \{\{0, n, 2n, \ldots\}, \{1, n+1, 2n+1, \ldots\}, \ldots, \{n-1, 2n-1, \ldots\}\}$. Elements of $\mathbb{Z}_n$, often called "modular numbers", 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 provide here the minimum amount of TLS information necessary to understand this paper. We refer the reader to [10, 2, 24, 18] for a more comprehensive TLS perspective. TLS exploits code regions that dynamically expose a 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 assigned to 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 state from the global state: each thread buffers its write-accesses, and commits them when it becomes master. It follows that WAR and WAW dependencies are implicitly satisfied. *In-place* models [16, 8, 20] access (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 (π) that defines how iterations are assigned to threads so likely dependencies are satisfied. $\pi : \{0, \ldots, W-1\} \rightarrow \{0, \ldots, C-1\}$ 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 π, 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 ($|\mathtt{k}_i - \mathtt{k}_j| < C,\ 0 \leq i, j < C$, where $\mathtt{k}_i$ 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 $j + 4\mathbb{Z} \equiv \{j, j+4, j+8, ...\}$ execute on thread j. We observe that, with this code, we can only partially exploit the available hardware-parallelism: we use 4 threads although we have 8 processors. To increase the iteration granularity, we can 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 $\{0, 4, 8, 12\}$.) It is worth noting however that this

way of increasing iteration-granularity is only applicable to *in-place* TLS models. (For a serial-commit model, the serial write-back phase cannot be achieved in any effective way, and hence $W = 4 =$ the cross-iteration dependence distance.)

The second example takes $B(\texttt{i})$ to be `a[i] = a[i] + 2`, and hence corresponds to (cross-iteration) dependency-free code. Without considering the iteration-granularity aspect, 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 $\{0, 1, 2, 3, 32, 33, 34, 35, ...\}$ 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 `LdVct[]` is a vector with as many entries as the size of an array `arr[]` that requires speculative support. A read from `arr[i]` in iteration `r` sets `LdVct[i]=r` *iff* `r` is currently the maximal iteration that has read `arr[i]`. A write to `arr[i]` by iteration `w` *discovers* a RAW *violation* when `w<LdVct[i]` since iteration `LdVct[i]` should have read the value written by `w`, but it did not.

To decrease speculative storage (`LdVct`) size, a (not one-to-one) hash function, of form $\texttt{hash}_{s,q,Q}(x) = ((x - s)\ \texttt{quo}\ q)\ \ \texttt{rem}\ Q$ can be used to map memory locations into indexes in `LdVct`. Now, the data space is partitioned into equivalence-classes ($x_1 \sim x_2 \Leftrightarrow \texttt{hash}(x_1) = \texttt{hash}(x_2)$), 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) `hash`, good speed-up is achieved only when the number of false-positives ($\texttt{hash}(x_1) = \texttt{hash}(x_2)\ \&\ x_1 \neq x_2$) 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 `hash`es will likely translate to poor performance.)

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

3}, $j \in \{0,..,7\}$, yields `hash`$(x) = ((x - s)$ `quo` $16)$ `rem` 8. One can verify that thread i accesses addresses that map to i via `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 `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, classical dependence analysis is likely to indicate that loops must be executed sequentially, 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 ($x - y = c\ (modM)$ 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

solved, 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; $\texttt{mode} : \texttt{SPP}_{dom} \rightarrow \{\texttt{r}, \texttt{w}\}$ represents this relation. We denote by $\texttt{A}_{dom}$, $\texttt{IS}_{dom}$ and $\texttt{SPP}_{dom}$ the domains of valid addresses, loop iteration space, and SPP. Hence $\texttt{A}_{dom}$, $\texttt{SPP}_{dom}$ and $\texttt{IS}_{dom} \subset \mathbb{Z}$, where we consider iteration i to be the i^{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 $\text{PIA}_{\texttt{q}} \subset \{\ (a, i) \mid a \in \texttt{A}_{dom}\ \texttt{and}\ i \in \texttt{IS}_{dom}\}$, $\texttt{q} \in \texttt{SPP}_{dom}$.

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 $(a, i_1) \in \texttt{PIA}_{\texttt{q}_1}$, $(a, i_2) \in \texttt{PIA}_{\texttt{q}_2}$, $i_1 < i_2$, $\texttt{mode}(\texttt{q}_1) = \texttt{w}$, and $\texttt{mode}(\texttt{q}_2) = \texttt{r}$, then we have a true-dependence (RAW) with the source being executed in iteration $\texttt{i}_1$ and the sink in iteration $\texttt{i}_2$. This leads to the following definitions:

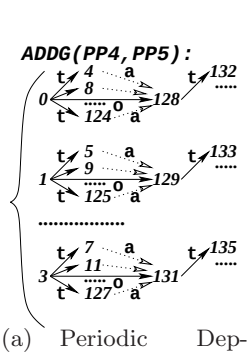
Definition 1 (Dependency-Class Notation). *We denote by (it_{src}, it_{snk}, tp) the class of run-time, cross-iteration dependencies, such that the source/sink of the dependency (on some memory location) is executed by the iteration numbered it_{src} / it_{snk}, respectively, and $tp \in \{t, a, o\}$ 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).*

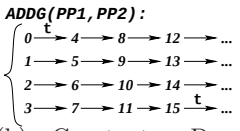
```
const int D = 4; const int B = 128;
1 for(int i=D; i<N; i++) {
2   a[i]    = ....  ;          // PP1
3   ....    = a[i-D];          // PP2
4   if (i%8 = 1)
5     ....  = a[i-1];          // PP3
6   a[i%B]  = ....  ;          // PP4
7   ....    = a[i%D];          // PP5
8   if(highlyUnlikelyCond())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

```
| if(cond1 || cond2) {
|        ... // PP1
| } else {
|        ... // PP2
| }
|
| BECOMES
|
| if(cond1) {
|        ... // PP1
| } else if(cond2) {
|        ... // PP1'
| } else ... // PP2
```

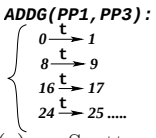
Fig. 1. A. Motivating Example B. Branch Normalization



(a) Periodic Dep-Pattern



(b) Constant Dep-Distance



(c) Scattered Dep-Distance

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

After (just-in-time) profiling a number of iterations, we construct for each SPP pair ($(q_1, q_2) \in$ PPP) their associated directed acyclic dependency graph ($\text{ADDG}_{(q_1,q_2)}$), 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(a) (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 `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 $\texttt{N} \gg W_{max}$ 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 $i, i+4, i+8, ...$, where $0 \leq i < 4$. This satisfies the dependencies in Figures 2(a) and 2(b), while light synchronization is introduced to satisfy the dependencies between threads 0 and 1 (those in Figures 2(c)).

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* $<M>$ *and the* step *operator* $|M>$ *and of characteristic* M *for element* (a, b) *as:*

$$(a,b)<M> = \{(x,y) \mid x \equiv a \ (mod\ M) \ \texttt{and}\ y \equiv b \ (mod\ M)\}$$

$$(a,b)|M> = \begin{cases} \{(a+kM, b+kM) \mid k \in \mathbb{N}\}, \texttt{if}\ a < b, \\ \{(a+kM, b+(k+1)M) \mid k \in \mathbb{N}\}, \texttt{if}\ a \geq b \end{cases}$$

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

Note that the step operator is more precise than the modulo operator: $S|M> \subseteq S<M>$. For example $(0,8) \in (0,0)<4>$ but $(0,8) \notin (0,0)|4>$. We represent the ADDG in Figure 2(c) as $(0,1)|8>$, 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(a) via the modulo operator as $\cup_{0\leq i<4}(i,i)<4>$ (the step operator fails to represent it since, for example, $(0,8) \notin (0,0)|4>$). Furthermore, the ADDG in Figure 2(b) also requires the modulo operator $(\cup_{0\leq i<4}(i,i)|4>)$ 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: $(a_1+b_1\mathbb{Z}) \sqcup (a_2+b_2\mathbb{Z}) = a_1 + \texttt{gcd}(b_1,b_2)\mathbb{Z}$ if $\texttt{gcd}(b_1,b_2)|(a_2-a_1)$ 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 $g = \texttt{gcd}(m,M)$, 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: $(a,b)|m_1> = \{(a+k*m_1, b+k*m_1) \mid k \in \mathbb{Z}\} \subseteq \{(a+e+k*m_2, b+e+k*m_2) \mid k \in \mathbb{Z},\ e \in [m_1]_{m_2}\} = \cup_{e\in[m_1]_{m_2}}(a+e,b+e)|m_2>$.

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

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: $\{(0,1)\}<8> \cup \{(0,1)\}<18> \subseteq \{(0,1)\}<2>$, 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: $\{(0,1)\}|8> \cup \{(0,1)\}|18> \subseteq S|18>$, 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] mod` 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 $S|m>$, and $g = \texttt{gcd}(M,m)$. The smallest set $U', U \subseteq U'$, of the form $S'|M>$ exists and is obtained when: $S' = \{(x,y) \mid x \equiv a + e$ `and` $y \equiv b + \alpha * m + e$ `mod` M, `where` $(a,b) \in S, \alpha = 0$ `for` $a < b$ `and` 1 `otherwise`, `and` $e \in [g]_M\}$ For the modulo relation the set S' can also be computed, but $S'<M> \equiv S<g>$.*

Proof. Straightforward application of the finite additive subgroup theory.

The rest of this section gives the unification rules for the step and modulo relations (sets). We introduce first the *degree of parallelism* of a set under the modulo/step form, which is useful in simplifying the congruence formulas. Intuitively, upon unification, we aim to obtain the congruence formula that yields the 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 $S<m>$, where only iterations $0..(m-1)$ are considered. The* degree of parallelism *of $S<m>$ is $\lceil m/l_{max} \rceil$. The same holds for $S|m>$.*

Definition 6 (Step/Modulo Unification).
MS.1: $S_1|m> \sqcup S_2<m> = (S_1 \cup S_2)<m>$
MS.2: $S_1|m_1> \sqcup S_2<m_2> = ([S_1|m_1>]_m \cup [S_2<m_2>]_m)<m>$, *where* $m \in \{m_2, \texttt{gcd}(m_1, m_2)\}$ *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 Basic Patterns

Figure 2 identifies three dependency patterns that may not necessarily prevent parallelism from being extracted, and hence constitute the basic building-blocks of our analysis. We expect that each ADDG associated to a certain PPP falls in one of the three categories. Combining among ADDGs corresponds to unifying set-congruence formulas as described in the previous section, and yields an algebra of patterns. This section is not intended to present the exact pattern-matching algorithms we use, as the paper does not aim to make a contribution in this direction. We restrict to (i) asserting the main pattern characteristics such an algorithm should identify, (ii) giving the pattern's congruence formula, and (iii) where not already discussed, presenting how formula unification is achieved.

We also note that we allow a few[4] dependencies to fall outside our patterns; they are tolerated as rare events, which is consistent with our two-sided analysis. These dependencies are stored inside a per-ADDG residue set, and are taken into account when the decision is made of whether TLS application is effective or not.

Repetitive Scattered Dependencies Pattern: The first basic pattern corresponds to the one shown in Figure 2(c). Its defining properties are: (i) there are very many connected components, each containing a small number of nodes, l (typically 2), (ii) the difference between the corresponding nodes of two consecutive connected components, denote it by m, is constant and equal between pairs of corresponding nodes (formula's characteristic), and (iii) the degree of parallelism $\texttt{ParDeg} = \lceil m/l \rceil$ 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 pattern's formula is $S|m>$,

[4] We use a 5% threshold of the per-ADDG total number of dependencies.

where $S \subseteq \mathbb{P}(\mathbb{Z}_m \times \mathbb{Z}_m)$ 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(c) formula is $(0,1)|8>$.

Constant Dependency Distance Pattern: The second basic pattern corresponds to the one shown in Figure 2(b). Its defining property is that (i) it consists of several l connected components, (ii) where consecutive nodes i and j on the same component satisfy the invariant $i - j = m$, with m constant and $l \leq m$. Denoting by R the set containing the roots of the connected components modulo m, the pattern's formula is $\{(i,i)|i \in R\}<m>$. For example, the ADDG in Figure 2(b) formula is $\{(i,i)|0 \leq i < 4\}<4>$.

Dependency-Free Window Pattern Although the ADDG in Figure 2(a) can be represented via the $\{(i,i)|0 \leq i < 4\}<4>$ formula[5], it corresponds to a more general, orthogonal pattern. Its defining property is that there are several nodes (nodes $[0,3]<128>$) that are the source/sink of many dependencies, and the structure is repetitive. If these iterations are executed sequentially, the remaining iterations ($[4,127]<128>$) cause no dependency-violations, hence they can be executed in parallel, out of order. The semantics is that for every 128 iterations, we apply a (synchronization) barrier and execute the first four iterations sequentially. The pattern can be identified by recursively eliminating the node featuring the highest number of incoming/outgoing edges from the graph; the pattern holds if eliminating few nodes results in only singleton nodes. Space constraints prevent us from formalizing this pattern here.

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 (sort the per SPP addresses; constructing the ADDG for a PPP is then linear). This is so because in practice we do not have to analyze each PPP, but, roughly, only those that refer to the same variable – we can determine the maximal and minimal addresses accessed at each SPP and construct ADDGs only between SPPs whose addresses overlap. Furthermore, pattern-matching ADDGs to determine formulas should be (in worst case) linear in the number of ADDG's dependencies, while unifying formulas among ADDGs (PPP) under the presented algebra is cheap.

The order in which (ADDGs) formulas are unified is important: a state-of-art framework would aim to assign iteration to threads such that most dependencies are resolved, while preserving the optimal degree of parallelism (P). Other PPPs whose unification would yield too conservative results are (lightly) synchronized (see Section 3.2). This paper does not discuss these heuristics.

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.

[5] If $\mathtt{D} \nmid \mathtt{B}$ – see Figure 1, the formula does not hold, but parallelism can still be extracted

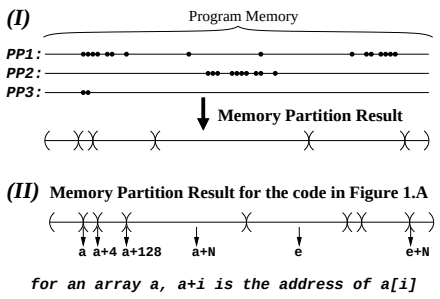


Fig. 3. Illustrating Memory Partitioning

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). Partitioning memory in this fashion provides opportunities to adapt the TLS model behavior to the access-patterns of each variable. Section 4.2 presents a fine-grained memory partitioning by which each individual TLS model's hash functions are computed.

4.1 Building Variable-Based Memory Partitions

Figure 3.*(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.*(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

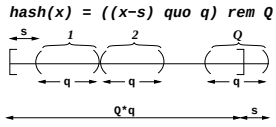


Fig. 4. Hash Function – Graphical View

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[6].

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[7] 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: $\texttt{hash}_{s,q,Q}(x) = ((x - s)\ \texttt{quo}\ q)\ \texttt{rem}\ Q$, 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: $a_1 \sim a_2 \Leftrightarrow \texttt{hash}_{s,q,Q}(a1) = \texttt{hash}_{s,q,Q}(a2)$. Figure 4 graphically depicts this view: there are Q intervals of equal length q. All the addresses a such that $(a - s)\ \texttt{rem}\ (Q * q)$ belong to the i^{th} interval and are in one equivalence class. This class corresponds to the union of cosets: $(i * q) + Q\mathbb{Z} \cup (i * q + 1) + Q\mathbb{Z} \cup \cdots \cup (i * q + q - 1) + Q\mathbb{Z}$. The use of the offset s is to align equivalence-classes in $\texttt{hash}_{0,1,Q*q}$ so that they can be safely collapsed by interval formation into $\texttt{hash}_{s,q,Q}$.

Intuitively, `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

[6] I.e. use a one-to-one mapping of addresses to speculative storage: those memory locations are already likely to generate dependency-violations – we should avoid increasing that probability by introducing false-positives.

[7] 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)

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 Example and Speed-up Results

Some trivial examples of hash functions for TLS models have been presented in Section 2.3. Running our analysis on the Fast-Fourier-Transform application, succinctly presented below (`dual` takes increasing powers of 2 in an outer loop):

```
for(a=1; a<dual; a++)
  for(b=0; b<n; b+=2*dual) {
    int i=2*(b+a), j=2*(b+a+dual);
    x[j] = Exp(x[j+1], x[j]);  x[i] = Exp(x[i+1], x[i]); ... }
```

yields $\texttt{hash}(x) = (x \texttt{ quo } 8) \texttt{ rem } 4$ (w.r.t. accesses of the `x` array), where we have profiled for `dual>4` (power of 2), and assumed a 4-processor machine. (In general the maximal degree of parallelism is `dual`.) Hence, a vector with only 4 elements is needed to track dependencies. Our result is more general than that obtained via Masdupuy's (static) trapezoid analysis which discovers only that indexes α used with array `x`, satisfy: $\alpha - 2 * a \equiv [0, 1] \texttt{ mod } 4$. The latter allows parallelism on only two processors. The more conservative result is a consequence of the fact that an abstract interpretation framework is bounded to discover congruences modulo the first value of `dual`, which is 2, while we can profile for a conveniently large `dual` and have `dual` degree of parallelism.

Finally, 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 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 returns the value stored into a memory location while writes cause a rollback followed by a sequential fix-up. SpRO is very effective on coarse-grained memory partitions

that are mostly read and rarely written, since its read overhead is nearly 0 in both time and space. SpLIP's optimized `hash`es are computed as described in the previous section.

The column `OptHash` refers to the case when only SpLIP is employed, however on optimized `hash`es. 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. This still requires the coarse-grained partitioning, otherwise yielding too many false-positive rollbacks. Finally, the last column uses one-to-one `hash`es: `q=s=0` and `Q` is roughly the data-space size. (For `FFT` and `NeuralNetFW` columns 5 and 6 use roughly the same Q, i.e. speculative memory overhead.)

The differences between optimized and (i) naively chosen and (ii) one-to-one `hash`es 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 hence 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 `hash`es do not. One interesting 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 present an algebra for partitioning the iteration-space to threads such that repetitive dependencies are resolved and rare dependencies are ignored. Second, we use 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 have achieved this 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 *International Symposium on Memory Management*, 1998.
5. M. Cintra and D. R. Llanos. Toward Efficient and Robust Software Speculative Parallelization on Multiprocessors. In *PPoPP'03*, June 11-13 2003, San Diego, California.
6. F. Dang, H. Yu, and L. Rauchwerger. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *Proc. Int. Par. and Dist. Proc. Symp. (IPDPS'02)*, 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. *ACM TOCS*, May 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. W. Kim, R. E. Chong-Liang Ooi, B. Falsafi, and T. N. Vijaykumar. Reference Idempotency Analysis: A Framework for Optimizing Speculative Execution. In *PPOPP'01 Proceedings*. ACM, 2001.
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. Min. 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'08, available at www.cl.cam.ac.uk/∼co280/IWMSE/TLSlib.pdf*.
18. P. Rundberg and P. Stenstrom. An All-Software Thread-Level Data Dependence Speculation System for Multiprocessors. *The Journal of Instruction-Level Parallelism*, 1999.
19. Y. Sazeides and J. E. Smith. The Predictability of Data Values. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 248–258. IEEE Computer Society, 1997.
20. T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, S. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. In *PLDI'07*, 2007.
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 Proceedings*. ACM, 2002.

24. C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Micro-35 Proceedings*. ACM, 2002.