Loopapalooza: Investigating Limits of Loop-Level Parallelism with a Compiler-Driven Approach

DOI:
10.1109/ISPASS51385.2021.00030

Document Version
Accepted author manuscript

Loopapalooza: Investigating Limits of Loop-Level Parallelism with a Compiler-Driven Approach

Ali Mustafa Zaidi[1], Konstantinos Iordanou[2], Mikel Luján[2], and Giacomo Gabrielli[1]

[1]Arm Research
{*ali.zaidi, giacomo.gabrielli*}*@arm.com*
[2]Department of Computer Science, University of Manchester, UK.
{*firstname.surname*}*@manchester.ac.uk*

***Abstract*—Improving sequential performance of out-of-order processors is becoming harder. Further improvements may require exploitation of thread-level parallelism, on top of ILP, as it can provide better design and performance scaling. Unfortunately, previous "speculative multithreading" approaches have shown small gains and/or incur a high cost, particularly for general-purpose, non-numeric applications.**

This paper investigates the fundamental limits to sequential performance scaling through speculative multithreading — we present an LLVM compiler-driven limit study framework that investigates the limits of loop-level parallelism at run-time. This new study of loop-level parallelism demonstrates the potential for up to 4.6x and 7.2x geometric mean speedup on SpecINT2000 and SpecINT2006. Thanks to the additional consideration of recent parallelization schemes, such as generalized DOACROSS (HELIX), these potential speedups are higher than reported by previous state-of-the-art limit studies.

Our analysis further categorizes the various inter-thread dependencies and ordering constraints with respect to the specific architectural choices and techniques each would require for implementation. We then evaluate the relative importance of each such constraint for different application (benchmark) types, and provide insight into the cost/benefit trade-offs when designing systems for efficiently implementing speculative multithreading. Such insights should help the design of bespoke systems for speculative multithreading while achieving better speedups, efficiency, and scaling, relative to typical approaches which, thus far, have relied upon adapting conventional multi-core systems.

***Index Terms*—Speculative Multithreading, Thread Level Speculation, Loop level parallelism, Auto-parallelization, LLVM**

I. Introduction

Out-of-order CPUs employ increasingly complex techniques to extract instruction level parallelism (ILP) from a single thread to continue scaling sequential performance. However, with the end of Dennard scaling [1], and a slowing Moore's Law [2], it is not possible to offset the quadratically increasing resource and energy requirements of evermore aggressive ILP techniques by transitioning to smaller process nodes. There are now two major constraints when attempting to improve single-thread performance.

- **Complexity Wall:** Pollack's Rule [3] observes that a linear scaling of $O(n)$ in sequential performance incurs an $O(n^2)$ increase in CPU complexity, and thus energy and resource requirements. Typically this increased complexity was offset with the shift to a newer, cheaper-per-transistor process node, but with Moore's Law slowing [2] architects must accept ever-diminishing performance returns from quadratically more expensive designs.
- **ILP Wall [4]:** Even if unlimited microarchitecture scaling were possible, there is a fundamental limit to how much ILP is available to extract from a single thread (approx. 7 instructions per cycle (IPC) for non-numeric code).

Previous work has investigated *Speculative Multithreading (SpMT)*, or *Thread Level Speculation (TLS)*, as a means of overcoming both the ILP and complexity walls [23]. SpMT/TLS attempts to exploit an orthogonal, coarser-granularity of parallelism beyond ILP by partitioning a single thread into multiple threads of execution. As early as 1992, Lam *et al*. [6] observed that such exploitation of thread-level parallelism (TLP) across multiple-flows of control can expose up to an order-of-magnitude greater parallelism.

Unfortunately, SpMT/TLS research has not been able to practically materialize these hoped-for, order-of-magnitude speedups. While some gains are observed for numeric applications, non-numeric applications either see very limited gains, or otherwise incur substantial inefficiencies [7]–[14]. These challenges stem from the fact that these studies attempt to repurpose conventional coherent multi-core architectures for finer-grained SpMT – something they were not designed for.

The best speedups for integer applications are demonstrated by research projects that redesign a scalable architecture from the ground-up specifically for SpMT/TLS [16], [17], or otherwise enhance a conventional multicore with support for low-latency inter-thread dependency propagation [15]. Yet despite these advantages, and supported by specialized SpMT compilers, these attempts too exhibit inadequate scaling efficiency, e.g. with speedups of 19x on 64 cores (for the most amenable benchmarks) [17], or 6.5x on 16 cores for SpecINT2006 [15].

The practical challenges of designing an efficient, scalable SpMT/TLS architecture – one that achieves high speedups without incurring 2-3x greater area/energy/complexity – requires an in-depth understanding of the key limitations for SpMT/TLS performance scaling. We consider such key limitations to be rooted in the inter-thread dependencies and ordering constraints that must be efficiently addressed.

This paper investigates the limits of TLP in the context of loop-level parallelism. Each loop iteration is partitioned into an individual thread of speculative execution. We then

categorize the variety of inter-thread dependencies that must be preserved in order to maintain correct sequential execution semantics. Each category of dependency type may need to be handled differently, and the relative impact of each category on achievable speedups in the limit can guide the amount of architectural/system-level design/resource effort justifiable. Based on this analysis, we are then able to order the relative importance of various proposed architectural, compiler, software, run-time, and system-level features. That order would be required to maximize the TLP speedup potential when designing a pragmatic SpMT/TLS system. This paper makes the following contributions:

- We present a categorization of inter-thread dependencies across speculatively generated threads (Section II). This categorization highlights the distinct ways these dependencies must be managed across the system, both in the compiler, and in the execution architecture.
- We develop a loop-level parallelism analysis infrastructure that evaluates the potential limits of parallelism achievable (Section III). This infrastructure, Loopapalooza, can be configured to evaluate the impact of addressing each of the categories of dependencies through potential architectural techniques or parallelization strategies, and appropriately estimate speedups in the limit accordingly.
- Based on our infrastructure, we present an analysis of the impact of various parallelization schemes, and acceleration of various dependency types, on the achievable speedup for a broad class of application types. We indicate the relative importance of addressing each type of dependency to achieving maximum parallelization for each class of application (Section IV).
- Based on our analysis, we highlight from prior-art, various architectural and compiler proposals that might be well suited to efficiently addressing the most important of these speedup-limiting dependency types.

Although we focus our experimental study on TLS for loop-level parallelism, the analysis of inter-thread dependencies applies also to broader techniques such as function-call/continuation level TLS, or more general TLS as applied to 'control quasi-independent regions' [18], as all of the dependency categories would still apply.

II. Categorization of Ordering Constraints

The goal of speculative multithreading is to allow for parallel execution of 'control quasi-independent regions' (CQIRs). CQIRs are code regions with control-flow that are highly likely to execute in succession, such as iterations of a loop, or a function call and its continuation [18]. Running CQIRs as independent threads allows exploitation of TLP orthogonally to any ILP that may be exploited within each thread.

However, while these regions may be quasi-independent in terms of control-flow, there may be substantial data (RAW), and false (WAR, WAW) dependencies between them, through both registers and memory, that must be preserved for correctness. Furthermore, speculative execution must also preserve the sequential *observable semantics* of the program by ensuring any other I/O or side-effects occur only in the strictly-sequential program order intended by the programmer, as well as respect any architectural constraints on execution order, such as those imposed by the memory consistency model.

Table I summarizes the various ordering constraints and Loop-Carried Dependencies (LCDs) that restrict parallel execution of loop iterations. This section discusses the limitations on relaxing these constraints to expose parallelism. We split dependencies not only by their nature (True, False, Structural), but also whether they manifest through 'registers' or memory. The primary distinction between the latter two is that the occurrence of register dependencies is known at compile-time, whereas memory dependencies are assumed to only manifest at run-time. As the name suggests, register LCDs are typically mapped to CPU registers, whereas memory LCDs manifest through load/store instructions.

We further subdivide these LCDs into 'frequent' and 'infrequent' types, with the former indicating that the dependency occurs each iteration, while the latter indicating that occurrence is rare, perhaps due to infrequent address aliasing, or a rarely-taken control-flow path through the iteration.

A. True Static Loop-Carried Dependencies (LCDs)

We classify True Static (Register) LCDs for loops broadly into 'computable' and 'non-computable'. Computable register LCDs are those for which a compiler analysis can determine a static, compile-time known scalar evolution expression (SCEV). Examples include induction variables (IV), and mutual induction variables (MIV), but also, broadly any register LCD for which we can express its per-iteration value solely as a function of the iteration count. Thus, although, these are true LCDs, they are not considered a constraint on parallelization, as their value within each iteration of a loop can be generated thread-locally based on an iteration index. As described later in Section III, we rely on Scalar Evolution analysis [19] of LLVM to identify computable register LCDs.

All remaining register LCDs, for which we cannot find a SCEV expression, are then classified as non-computable register LCDs. However, as most register LCDs (for all non-trivial loops) are expected to fall in this category, it is necessary to attempt further classification of these, depending on what options may be available to accelerate them. These are further subdivided into the following categories:

- **Reduction Accumulators:** these have a recognizable, exclusively reduction/accumulation pattern across the loop iterations. While the updated values in each iteration may not be fully determined at compile-time, the update pattern itself is bounded and well understood.
- **Predictable Register LCDs:** although not compile-time computable, these LCDs are predictable at run-time through simple and known value prediction schemes.
- **Unpredictable Register LCDs:** All remaining register LCDs fall into this category.

Reduction LCDs manifest as read-modify-write operations in each iteration, but unlike a general non-computable LCD,

TABLE I: Various ordering constraints and dependencies restricting execution of loop iterations in parallel.

Dependencies			
Type Category		Frequency of Occurrence	Classification
TRUE	Register RAW	Frequent	(Mutual) Induction variables (IVs & MIVs)
			Reduction accumulators
			Non-computable and unpredictable register LCDs
		Infrequent	Non-computable but predictable register LCDs
	Memory RAW	Frequent	Compile-time known memory LCDs
			Frequent dynamically aliasing memory LCDs
		Infrequent	Compile-time known LCDs under infrequent control-flow paths
			infrequent dynamically aliasing memory LCDs
FALSE	Register WAW & WAR	Execution model and architecture dependent	
	Memory WAW & WAR		
STRUCTURAL	Register (SP)	Frequent for loops containing function calls	
	Memory (Stack)		

they follow a very specific accumulator-like pattern, and thus may be 'decoupled' from the remainder of the execution of the loop – removing the reduction as serializing dependency between iterations, but implementing it instead as an accelerated operation off the critical path of the loop. Tree or linear-chain based reduction functionality (for associative and non-associative reductions, respectively) can be added to SpMT architectures. Such techniques would gather the intermediate values from parallel iterations and reduce them independently. Examples of such hardware functinality can already be found in advanced SIMD/Vector architectures, e.g. Arm SVE [20].

Our second category of non-computable register LCDs relies on the fact that though a register LCD may not have a compile-time knowable SCEV, it may still evolve highly predictably at run-time. Thus, in cases where we can assume the presence of a run-time data value predictor targeting such LCDs [18], for every instance of these that is correctly predicted, the dependency need not be considered a sequentializing dependency between iterations. If such a dependency is found to be highly predictable, we can classify it as an 'infrequent' register LCD, given that its manifestation as a value-misprediction is rare, and depends on its run-time behavior. Several previous works have investigated the application of value predictors to SpMT (see [23] for a survey).

All remaining Register LCDs are then finally classified under the true, frequent, unpredictable, non-computable LCD category. Unlike the other classes of register LCDs, this one cannot be assumed removed from the parallel loop execution critical path by the compiler or clever architecture. Thus it can only be handled in parallelization schemes that support per-iteration inter-thread data-flow synchronization such as DOACROSS, or HELIX (a generalized DOACROSS) [14], [22]. Hopefully, as we see improvements in compiler analyses (for SCEVs, and capture broader classes of reduction patterns), as well as more sophisticated value-prediction schemes, more such dependencies could be moved into one of the earlier categories, further improving opportunities for parallelism.

B. True Dynamic Loop-Carried Dependencies (LCDs)

Unlike Register LCDs, Memory LCDs cannot always be identified at compile-time. While pointer analysis may be applied to refine the confidence on when a memory RAW hazard may exist between any two iterations, typically such analysis either provides poor confidence, leading to requiring conservative synchronization to ensure correct ordering, or impractical levels of compiler effort [14]. Instead of points-to analysis, our limit study framework relies on dynamic conflict-tracking during execution in order to classify true dependencies through memory as either 'frequent' or 'infrequent'.

C. Frequency of LCDs and Parallel Execution Models

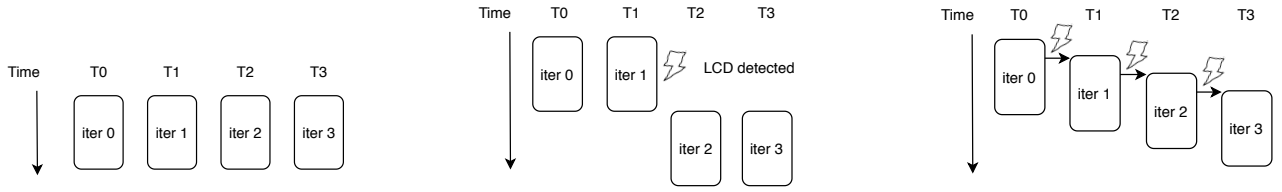
This classification of LCDs into 'frequent' and 'infrequent' matches with the available options for parallel execution typically considered for SpMT systems [23]. The execution models for SpMT studied hereafter are: DOALL, Partial-DOALL, and DOACROSS/HELIX.

(1) DOALL: All loop iterations may be started speculatively, and execute assuming no conflicts. If a conflict occurs, abandon parallel execution, and mark the loop as suitable for serial execution only (Figure 1a). This model does *not* support any memory LCDs or non-computable register LCDs.

(2) Partial-DOALL (PDOALL): Similar to DOALL, all loop iterations may be started speculatively. But upon conflict, the conflicting iteration thread and all younger iterations must abort and discard their speculative state. Parallel execution of this and all future iterations can however restart after the conflict is resolved (Figure 1b). This model only supports *infrequent* memory and register LCDs.

(3) DOACROSS/HELIX: All loop iterations may be started speculatively, but iteration execution also includes synchronization points where younger, consumer iterations must wait for older, producer iterations to signal that it is safe to proceed (Figure 1c). This allows all iterations to continue execution non-speculatively with all RAW dependencies, including frequent ones, correctly satisfied through synchronization.

HELIX is a generalization of the traditional DOACROSS approach, which only supports a single point of synchronization between iterations. Thus if multiple memory LCDs existed, DOACROSS synchronization would occur only after the last write in the previous iteration and immediately before the first read in the next iteration. HELIX instead allows support for multiple synchronization points, one for each distinct memory LCD that a compiler can isolate, thereby potentially exposing more parallelism [14], [15], [22].



(a) DOALL execution (b) partial-DOALL execution (c) DOACROSS / HELIX execution

Fig. 1: Parallel execution models.

Infrequent memory LCDs may be handled through speculative techniques that treat each thread as an 'ordered transaction' - much like transactional memory systems, each iteration/thread is treated like an executing transaction, and if a memory LCD violation occurs, the 'younger' thread(s) must restart to fix their misspeculated execution [5], [23]. The PDOALL model simulates such an execution environment.

Note that if the frequency of conflicts for an LCD is sufficiently high, the PDOALL approach would be unsuitable as it would provide little gain over serial execution, even if one ignores the overheads of thread spawning, commit, and conflict checking. Worse still, it would incur substantial energy overheads from the unnecessary speculative execution of all the threads that repeatedly get restarted.

Frequent LCDs are instead, better handled through non-speculative approaches, such as those proposed for the DOACROSS/HELIX [14], [15], [22], that rely on explicit synchronization between iterations to correctly order producer and consumer operations. This approach can also be used to support frequent non-computable register LCDs, for instance, by either lowering these to memory [14], or providing other architectural mechanisms to pass values between iterations at synchronization points [15]. Note that although such mandatory synchronization between iterations may reduce parallelism compared to the earlier parallel SpMT approaches, it allows us to support all frequent LCDs through both registers and memory, thus potentially enabling much broader *coverage* of loops that can now be parallelized.

D. False Loop-Carried Dependencies (LCDs) through Registers and Memory

While the RAW, or true memory dependencies are critical for program correctness, the WAW, WAR, and RAR ordering anti-dependencies must also be preserved in a real system to maintain correct observable semantics (WAW, WAR), as well as architectural memory consistency behaviors (WAR, RAR). However, these anti-dependencies need not manifest as sequentializing dependencies, particularly if we can separate the speculative execution of loop iterations from their commit.

As highlighted in [23], there are two main strategies for implementing data version management in an SpMT system: lazy and eager. *Lazy versioning* stores speculative state separately from the main program state, and the speculative state is *committed* to this main memory only upon successful *commit* of the speculative thread in the correct order of threads. And if the speculative thread is aborted, the speculative state is directly discarded without ever becoming observable by the rest of the system. *Eager versioning* on the other hand performs all speculative updates *in place* in the main program state, so all speculative updates are immediately visible to the whole system. A separate *undo log* of updates is also maintained by each thread to be used in case a roll-back of changes must be performed if a thread is aborted.

The advantage of eager versioning is that the process of committing a speculative thread has zero cost, where lazy versioning may incur a commit cost for marking state as no-longer speculative, or copying to main state. This may be beneficial when thread aborts are expected to be rare. Unfortunately, eager versioning makes all speculative changes observable out of order. Thus, if we wish to avoid any violations of a program's sequential observable semantics, while still utilizing eager versioning, we must categorize and treat all false dependencies as carefully as we would the true dependencies. Note that if an SpMT execution model is designed *without* the assumption of having implicit sequential execution ordering, or sequential observable semantics, as is the case with the *explicitly ordered tasks* model used by SWARM [16], [17], eager versioning remains a viable option.

For systems based on more conventional architectures, a simpler alternative may be to rely on lazy versioning instead, wherein the system effectively *decouples* execution of a thread from its commit (i.e. when its effects become *observable*). In-order commit ensures that all writes appear to occur in the correct program order, thus resolving WAW and WAR dependencies. This is analogous to how a typical out-of-order processor decouples execution of *instructions* from commit – the former is only restricted by the true dataflow dependencies, while the latter must occur in correct program order.

For simplicity, we assume in the remainder of the paper that all false dependencies are handled appropriately through lazy versioning, coupled with in-order commit of thread state.

E. Structural Loop-Carried Dependencies (LCDs)

Another machine model artifact that can restrict concurrency is the call stack, for cases where a loop iteration being parallelized may call a function. Assuming the function being called is itself parallelizable (e.g. read-only or pure, thread-safe, or infrequently conflicting with subsequent calls in future iterations), it may be beneficial to instantiate multiple copies of a function as its invoking loop iterations execute

concurrently. Unfortunately, due to the sequential, LIFO nature of conventional call stacks, updates to the stack pointer are required to occur in sequential program order, and the same set of call stack locations are expected to be used by each of these function calls, even though across distinct calls these are neither true nor false dependencies as classified above.

Explicitly parallel, work-stealing run-times typically employ 'cactus-stacks' [24] in order to overcome this problem, as these allow for disjoint stack space allocations for concurrent instances of function calls. While, the implementation of such cactus-stacks is an part of the parallel user-space work-stealing run-time that manages *explicit* parallel execution, it is possible to presume that similar principles may be adapted for an *implicitly* parallel SpMT system without requiring such a run-time - by treating both the stack pointer and allocations to the stack frame as part of the speculative, *transactional* state of an uncommitted iteration, it would be possible to disambiguate between concurrent instances of the stack frame created by parallel iteration-threads, while preserving sequential observable semantics through in-order commit, and without causing conflicts due to this particular structural hazard.

Thus again, we assume reliance on lazy versioning for speculative state and in-order commit, together with special consideration for stack frames, in order to enable parallelization across function-calls in our study. An alternative approach is to use compiler-directed heap-based allocation of function-local data, as used by MIT SWARM [16], [17], which again is permissible due to their explicitly parallel task ordering model and relaxed expectations about sequential semantics.

III. Loopapalooza – The Limits Study Framework

Loopapalooza (**LP**) is not an auto-parallellization tool, but rather an analysis framework that estimates parallel speedups achievable over sequential code from combinations of the different parallel execution models and dependency constraint relaxations described in Section II. It achieves this by adding instrumentation to a sequential program at compile-time, to track loop entry, iterations, exits, as well as potential dependencies such as memory accesses, function calls, and register LCD values. To produce an estimate of the loop-level parallelism available for a given program, **LP** has a compile-time and a run-time component, described below.

A. Loopapalooza Compile-time Component

Unlike previous limit studies on parallelism, LP takes a compiler-driven approach by leveraging static analysis as much as possible, in this case from the LLVM framework, in order to help categorize dependencies across loop iterations. The compile-time component takes as input the LLVM intermediate representation (IR) of the compilation units after they have been optimized (using `-Ofast`). It then applies various transform passes to help simplify loop analysis. In particular, loops and induction variables are canonicalized using the *loopsimplify* and *indvars* passes; the canonicalization of loops is important to be able to uniquely identify loops within arbitrarily complex loop nests. LP then uses the *scalar evolution* (SCEV) pass to assist the categorization of static dependencies as computable (IVs and MIVs) or non-computable, and the *recurrence descriptor* from the *induction variable users* pass to detect reduction patterns.

These are then followed by custom instrumentation passes, developed specifically for **LP** to insert call-backs to the run-time component. These call-backs are used at run-time to keep track of the dynamic IR instruction count, to signal entry/exit to/from functions, to record loop entry, iteration, and exit boundaries, and to track addresses of memory accesses that can be involved in run-time dependencies. Based on the aforementioned SCEV and recurrence analyses, we separate the computable and reduction register LCDs, from the remaining non-computable LCDs. For configurations that evaluate the efficacy of value prediction for the latter, LP can then insert call-backs to track the per-iteration values they produce.

LLVM IR instruction counts per basic block are hard-coded at compile time into a call-back for each basic block. At run-time these call-backs allow us to track a total IR instruction count to represent a sequential run-time metric. The loop header, loop latch and loop exit call-backs can sample this running sequential IR cost counter to establish loop start, iteration start and iteration length time-stamps per iteration for each loop being tracked. The run-time can then use these time-stamps to derive additional counters 'simulating' a parallel execution cost for the DOALL scheme for each suitable instrumented loop. The call-backs tracking memory addresses are additionally used to simulate parallel costs for the PDOALL and HELIX-style approaches, by tracking and comparing time-stamps for conflicting accesses.

We use this combination of compile-time analysis and run-time instrumentation to ensure that all loop-carried dependency types described in Section II can be properly accounted for. The advantages of this compiler-directed approach are threefold. First, unlike more idealistic trace-based limit studies, using existing compiler analyses directly relates our results to what may be practically achievable by code generation when targeting prospective future SpMT architectures. Second, as the quality of such analyses continues to improve, and those improvements should be reflected in the overall achievable speedups. Third, by using compile-time analysis to filter out accelerated dependencies, dependencies statically proven not to occur, as well as non-parallelizable loops (for a particular configuration), the overheads of run-time dependency tracking, both in terms of execution time and memory footprint, can be minimized. This allows **LP** to scale to large applications.

B. Loopapalooza Run-time Component

The run-time component is a C++ library that is linked with the instrumented program, and implements the instrumentation call-backs inserted in the code by the compile-time component. It is primarily responsible for monitoring the occurrence of dependencies and for determining the overall speedup over sequential execution. The final cost, in terms of execution

time[1], of a loop is determined by the specific parallel execution model, selected among the models described in Section II, handles the conflicting iterations.

Under DOALL, when a conflict is detected across loop iterations, the loop is marked as sequential and the final loop cost is calculated as the sum of the costs of the individual loop iterations. If there are no conflicts across iterations, the loop is considered parallel and the final loop cost is the cost of the slowest iteration (see Figure 1(a)), as determined using the loop start, and the largest loop iteration time-stamp.

Under Partial-DOALL, when a conflict is detected, the loop is still considered a candidate for parallelization. However, when the number of conflicting iterations exceeds 80% of the total number of iterations, the loop is marked as sequential. Until a conflict is detected, the cost of the loop matches the cost of the slowest iteration as above for DOALL. Upon conflict detection, the cost of the slowest iteration encountered up to that point is added to the loop start (or completion of the previous parallel region) time-stamp. This becomes the new starting time-stamp for a new Partial-DOALL phase. The internal counters tracking the longest iteration are reset to again start tracking the slowest iteration in this new parallel phase. The tracking continues until a new conflict is detected again, or until the loop exits. In other words, as Figure 1(b) illustrates, when a conflict is detected under the partial-DOALL execution model, we resolve the dependency by delaying the start of the conflicting iteration to match the end of the slowest iteration from the previous conflict-free phase of execution.

The HELIX-style model can tolerate frequent LCDs between iterations by assuming the presence of synchronization among iterations. This synchronization is modeled by tracking two separate values. We record the cost of the slowest iteration to execute in the entire loop ($iter_{slowest}$). Then, assuming all iterations start in parallel at the same time-stamp, we also record the largest time-stamp delta between each producer and consumer for any manifesting LCD between iterations ($delta_{largest}$). Then, for a loop with num_{iter} iterations, the parallel loop execution cost is then calculated as:

$$HELIX_{time} = iter_{slowest} + delta_{largest} * num_{iter}.$$

Finally, loops for which $HELIX_{time}$ is greater than the serial execution time, we mark these as serial, and only record their serial execution time. Once the loop execution cost for the innermost loops is generated based on the execution model, it is then propagated up to the nest of parent loops and functions. Thus, these outer loops may also compute their parallel and non-parallel execution costs.

C. Loopapalooza – Value Prediction

LP supports emulation of value prediction for non-computable register LCDs. Currently four types of value predictors are supported: (a) last-value, (b) stride, (c) 2-delta stride, and (d) Finite Context Method [40]. These are integrated into the run-time component of **LP** and they can be used individually or combined together as a hybrid predictor that selects between the individual predictors based on confidence counters. For this study, we assume perfect hybridization, in that if any of our predictors correctly predicts an LCD value, we assume we have a correct prediction. The set of predictors may be expanded, and more realistic hybridization schemes can also be implemented to test their effects. But for this limit study, assuming perfect hybridization suffices to illustrate the sensitivity of SpMT to predictable register LCDs, while avoiding inaccuracy introduced due to a rudimentary or unoptimized hybridization scheme.

[1]Note that **LP** always takes the dynamic LLVM IR instruction count as the approximation of execution time for regions of code. Furthermore, as this is a limit study, we assume infinite resources are available, allowing for arbitrary degrees of parallelism, only limited by the defined LCDs.

TABLE II: Configuration flags and their definitions.

Flag	Definition
-reduc0	Reductions are treated as non-computable LCDs.
-reduc1	Reductions are considered parallel with no overheads
-dep0	Non-computable LCDs are not considered parallelizable.
-dep1	Non-Computable LCDs are lowered to memory (and treated as frequent memory LCDs).
-dep2	Non-computable LCDs are accelerated using 'realistic' value prediction.
-dep3	Non computable register LCDs are accelerated using *perfect* value prediction.
-fn0	Loops with any function calls are marked as sequential.
-fn1	Only user and library function calls identified by the compiler as pure (read-only with no side effects) are considered parallel.
-fn2	Function calls solely to -fn1 functions, thread-safe (re-entrant) library functions, and user functions that have been appropriately instrumented by LP to capture memory read- and write-sets can be marked as parallel.
-fn3	All function calls can be parallelized.

D. Loopapalooza – Limitations

The **LP** framework has some limitations in the estimation of the available loop-level parallelism. However, given the nature of the limit studies herein described, we believe that such limitations are reasonable and their impact on the observed trends is minimal. Firstly, the estimation of potential speedups from parallelization is based on dynamic LLVM IR instruction counts. That means no microarchitectural aspects are taken into account in the speedup calculation - this is appropriate for limit-study style first-order analyses. Secondly, there is no accounting for execution time spent in system calls and/or pre-compiled library calls which we are unable to instrument with our call-backs. However, for the considered benchmarks, the only pre-compiled libraries are the C/C++ standard libraries. Finally, parallelization overheads (spawn, schedule, sync, commit, etc.) are not taken into account, though this is in line with our goal to determine an upper bound for the amount of exploitable parallelism within loops.

IV. Experimental Results – Limit Study

We investigate the limits to loop-level parallelism under DOALL, Partial-DOALL and HELIX-style execution models for a variety of configurations and analyze the progressive

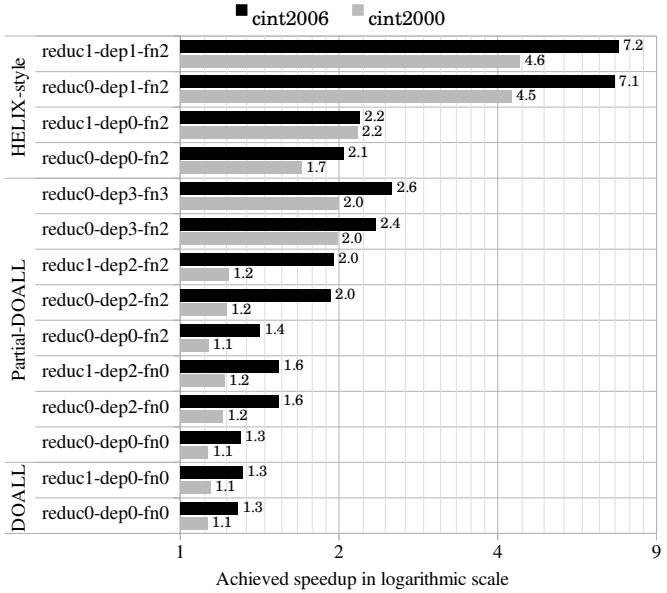


Fig. 2: GEOMEAN Speedups for *non-numeric* benchmarks (SpecINT 2000 & 2006) under various configurations.

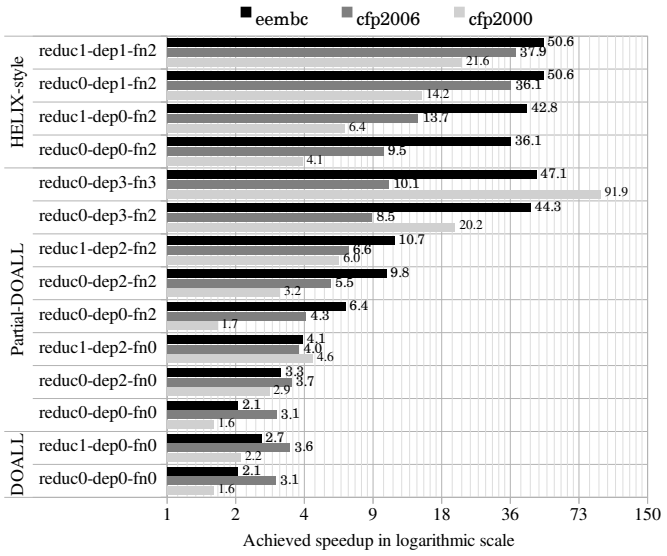


Fig. 3: GEOMEAN Speedups for *numeric* benchmarks (EEMBC, SpecFP 2000 & 2006) under various configurations.

gain in limit speedup as various parallelization constraints are gradually relaxed. We discuss the implications for potential implementations at each step. We group our benchmarks into *numeric* (EEMBC, SpecFP 2000 & 2006) and *non-numeric* (SpecINT 2000 & 2006), and Figures 3 & 2 present the results for these groups, respectively.

Table II summarizes the configuration flags used, with a brief definition for each. The minimum SpMT configurations attempted are the `reduc0-dep0-fn0` and `reduc1-dep0-fn0` for the DOALL execution model. For the former, reductions prevent loops from being considered parallel (`reduc0`), while loops with reductions are parallelizable in the latter (`reduc1`). Besides, reductions, non-computable register LCDs prevent parallelization (`dep0`), and so does the presence of any function calls (`fn0`).

Even with such highly restrictive configurations, we observe that numerical benchmarks (Figure 3) exhibit gains ranging from 1.6x–3.1x with `reduc0`, and rising to between 2.2x–3.6x with `reduc1`. In contrast, the non-numeric suites achieve much poorer gains, ranging from 1.1x–1.3x in the limit, even with infinite resources. Further relaxations of register LCDs (`dep1-dep3`) are incompatible with DOALL, as it does not support non-computable register LCDs, while further relaxations of function parallelization (`fn1-fn3`) provided no gains for both classes of benchmarks under DOALL.

To explore the criticality of infrequent register and memory LCDs to loop-level parallelism, we investigate various configurations for the Partial-DOALL (PDOALL) model. The minimum `reduc0-dep0-fn0` PDOALL achieves identical results to its DOALL counterpart for both benchmark classes, indicating that infrequent memory LCDs, which PDOALL supports, are not the bottleneck at this configuration, but rather non-computable register LCDs are. As we permit predictable, non-computable register LCDs to be parallelized with the `reduc0-dep2-fn0` PDOALL configuration, we observe a substantial increase to 2.9x–3.7x for non-numeric benchmarks, followed by another further increase to 4.0x–4.6x as reductions are also parallelized with `reduc1-dep2-fn0` PDOALL. Note that SpecFP2000 benefits greatly from both `reduc1` and `dep2`, while SpecFP2006 and EEMBC benefit more from the latter and some from the former. The `reduc0-dep2-fn0` and `reduc1-dep2-fn0` PDOALL configurations provide a more modest rise from 1.1x–1.3x to 1.2x–1.6x for non-numerical benchmarks, with `reduc1` having no effect.

Next, permitting parallelization across function calls (`fn2`) exhibits further substantial gains for the numerical benchmarks[2]. EEMBC benefits more from this as it performs even better with `reduc0-dep0-fn2` PDOALL than `reduc1-dep2-fn0` PDOALL, while the SpecFP suites show cumulative benefits of relaxing all three types of dependencies, with `reduc1-dep2-fn2` PDOALL. Of the non-numeric benchmarks, SpecInt2006 benefits from the `dep2-fn2` combination, rising from 1.6x to 2.0x, while SpecInt2000 sees only a marginal increase to 1.2x.

At this point, it is unclear whether parallelism in non-numeric applications is being restricted by the limitations of our value predictors (for `dep2`), the presence of non thread-safe function calls (`fn2`), or the presence of frequent LCDs (recall that PDOALL performs poorly in the presence of the latter). To investigate this, we attempt a set of unrealistic configuration options: `dep3` assumes we have a perfect value predictor, in essence, removing all non-computable register LCDs (including reductions) from limiting PDOALL parallelization, while `fn3` assumes all function calls are thread-safe. With these, we find that SpecInt2000 finally increase

[2]Results with `fn1` are ommitted, as this configuration is not unique; after parallelizing the call stack, we could parallelize all instrumented and thread-safe function calls. Table II includes `-fn1` to help define `-fn2`.

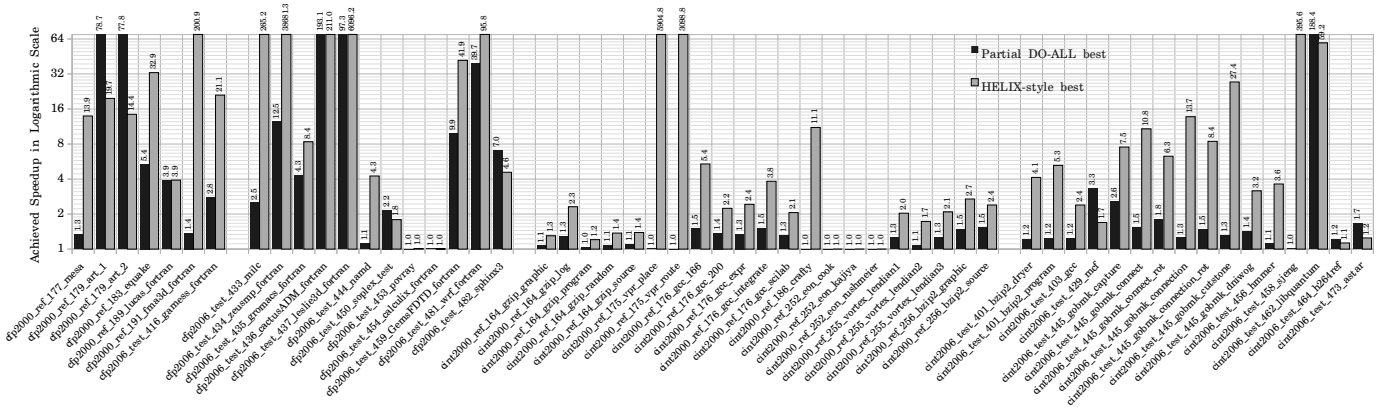


Fig. 4: All SPEC speedups for the best PDOALL (`reduc1-dep2-fn2`) and HELIX (`reduc1-dep1-fn2`) configurations.

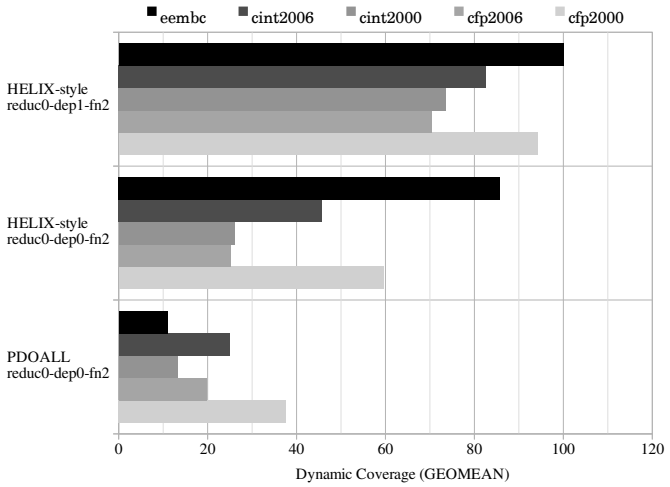


Fig. 5: *Coverage* for selected configurations.

from 1.2x to 2.0x, while SpecInt2006 increases from 2.0x to 2.6x for `reduc0-dep3-fn3` PDOALL. These are respectable gains, but the impracticality of implementing perfect value-prediction or perfect function call parallelization indicates that this may not be the best avenue to explore to accelerate such applications. The numeric benchmarks benefit even more from these relaxations, exhibiting 10x–92x gains for `reduc0-dep3-fn3` PDOALL.

To further extract parallelism from non-numeric applications, we must consider parallelism in the presence of frequent register and memory LCDs. The HELIX-style configuration natively simulates synchronization to handle frequent memory LCDs, while we combine HELIX with `dep1`, to support frequent register LCDs, by simply lowering these to memory. With frequent memory dependencies and function parallelization both supported under `dep0-fn2` HELIX, both non-numeric suites are able to achieve approx. 2.2x speedup, and with the addition of frequent register LCDs through `dep1-fn2` HELIX, we finally find respectable speedups for the non-numeric suites, with speedups of 4.6x & 7.2x for SpecINT2000 and SpecINT2006 respectively. The HELIX configurations also benefit the numeric suites, as they increase from 6.0x–10.7x for the best realistic PDOALL configuration (`reduc1-dep2-fn2` PDOALL) to 21.6x–50.6x for the bext HELIX-like configuration (`reduc1-dep1-fn2` HELIX).

Lessons Learnt — We observe that non-numeric benchmarks are sufficiently complex that their loops are serialized due to *frequent* true LCDs, both through memory and registers, as well as frequent structural (call-stack) hazards, due to the commonality of function-calls within loops. All of these dependencies are relaxed through `dep1-fn2` HELIX-style configurations, to unleash substantial parallelization potential. Note that the key factor in this gain is not strictly per-loop parallelism, but rather *coverage* in terms of of the amount of dynamic instructions that execute within parallel loops. Figure 5 illustrates the dramatic increase of this coverage as we transition from `dep0-fn2` PDOALL to `dep0-fn2` HELIX, and then to `dep1-fn2` HELIX. Recall from Amdahl's Law, that parallel speedup is a function of both degree of parallelism *and* fraction of code parallelized (i.e. coverage).

In contrast, numeric benchmarks are more balanced in their sensitivity to each class of ordering constraint highlighted by Table I. While they are more sensitive to *infrequent* register and memory dependencies, they still benefit from the increased coverage afforded by the HELIX style (Figure 5). Loop structure, control-flow and memory access patterns are mostly regular, and often compile-time predictable for this class.

Figure 4 shows individual benchmark performance across all of the SPEC benchmark suites for the best HELIX and PDOALL configurations. While the HELIX style approach provides a more consistent gain across the non-numeric benchmarks, there are a few cases where the PDOALL results are better, for instance, 179_art, 450_soplex, 482_sphinx, and 429_mcf. PDOALL is a speculative technique that only incurs a restart cost upon conflict, while HELIX is a non-speculative technique that imposes synchronization between all neighboring iterations irrespective of whether it is needed dynamically. Thus for loops parallelizable using either technique, it is quite likely that PDOALL would provide better per-loop speedups than HELIX, particularly if the inter-iteration conflict-rate is low. The consistently higher overall speedups achieved by

HELIX are more due to the much higher coverage enabled by this technique, and the fact that loops with high conflict rates incur higher restart costs with Partial DOALL than synchronization costs with HELIX.

V. Related Work

A review of auto-parallelization or TLS is beyond this paper and can be found in [23] [5] [55]. The most recent and comprehensive study of TLS performance potential appeared in 2010 by Ioannou *et al*. [47]. That study was able to simulate various TLS techniques, such as multiversioned caches, out-of-order thread spawning, dynamic dependency synchronization, checkpointing, and data and return value prediction. A trace-driven approach was used, and simulated an idealized 16-core system, considering options such as perfect value prediction, perfect dependency synchronization. A key observation was that significant potential for performance improvement exists mostly because of high coverage parallel inner loops (99%, 94% and 95% parallel inner loops coverage reported for SPEC CPU2000, CPU2006 and SPECJVM, respectively).

Furthermore, load imbalance and poor coverage were major factors limiting speedup, and thus effective task selection was significant for exploiting parallelism. In contrast, **LP** is compiler-driven instead of trace-driven, thus it can provide insights on the relationships between speedup, dependency categories, and coverage. Ioannou *et al*. [47] used 'perfect' value prediction, whereas we use specific predictors. **LP** also supports detailed parallel execution models for DOALL, PDOALL and HELIX. **LP**, however, distinguishes between limitations to parallelism and the challenges of effective run-time scheduling, focuses on the former, and does not consider aspects of the latter such as task selection and scheduling for bounded resources.

Kejariwal *et al*. [51] and [52] also explored the limits of TLP. They also considered overheads of threading, as well as the mispeculation penalties that occur from TLS. Their analysis claims that on SPEC CPU2000, TLS has a limited arithmetic mean speedup potential of 39.16% and a geometric mean speedup potential of 18.18% at the loop level. However, their analysis neither considered outer loops parallelization, nor out-of-order spawning of threads (nested parallelism).

Mentioned in previous sections are the SWARM [16], [17] and HELIX [14], [15], [22] projects. Both proposed frameworks for SpMT, and exhibit significant speedups. However, both achieved these gains in very different ways. SWARM proposed a novel architecture for executing explicitly ordered tasks, departing significantly from the conventional threaded von-Neumann model typically considered. With their *T4* SpMT compiler framework partitioning sequential code into a hierarchical nested tree of fine-grained light-weight ordered tasks (to allow exploitation of parallelism from multiple levels of a loop nest), they were able to achieve GEOMEAN 19x speedup on 64 cores, for benchmarks without any *frequent* LCDs. Unfortunately, SWARM cannot support frequent LCDs, so they offered no meaningful speedups for benchmarks with frequent LCDs (1.2x from 64 cores).

In contrast, HELIX supports frequent LCDs through inter-thread synchronization, and can parallelize outer loops, but does not support nested parallelism. Their first attempt [14] relied on helper-thread based prefetching to accelerate synchonization. Unfortunately, while it accelerated numeric benchmarks (12x from 16 cores), synchronization latency was too great, reducing coverage and thus speedup for non-numeric benchmarks (2.1x from 16 cores). Their follow-on work reduced synchronization latency by simulating a single-cycle ring-cache between neighbouring threads [15]. This substantially improved coverage and thus per-loop speedup for non-numeric benchmarks as well (6.5x from 16 cores).

Our **LP** framework simulates characteristics of both HELIX and SWARM/T4, supporting both frequent LCDs, and multi-level nested loop parallelization. The results confirm the importance of supporting frequent LCDs when parallelizing non-numeric benchmarks.

VI. Conclusions

This paper has investigated the fundamental limits to sequential performance scaling through speculative multithreading – we present a compiler driven limit-study framework that investigates the limits of loop-level parallelism at runtime. A distinguishing feature of our framework is the ability to evaluate the effects of various parallelization strategies, constraints and data dependencies. Our analysis categorizes the various inter-thread dependency types and ordering constraints with respect to the specific architectural choices and techniques each would require for implementation, evaluates the relative importance of each such constraint for different application types, and provides insight into the cost/benefit trade-offs when designing systems for efficiently implementing speculative multithreading.

This new limit study of loop-level parallelism demonstrates that it is possible to achieve as much as 4.6x and 7.2x speedup on SpecINT2000 and SpecINT2006 benchmarks. Thanks to our additional consideration of recent parallelization schemes, such as generalized DOACROSS (HELIX), these speedups are higher than reported by previous state-of-the-art limit studies.

These results are encouraging in terms of hidden parallelism in loops which is currently not exploited by either vectorization and/or finer-grain ILP extraction. The new insights should help the design of bespoke systems for speculative multithreading while achieving better speedups, efficiency, and scaling, relative to typical approaches which, thus far, have relied on adapting conventional multi-core systems.

Acknowledgments

We would like to thank Robert Kovacsics and Niall Murphy for their contributions and insights in the early stages of this work. Iordanou is funded by an Arm Ltd. & EPSRC iCASE PhD Scholarship. Luján is funded by an Arm/RAEng Research Chair award and a Royal Society Wolfson Fellowship.

REFERENCES

[1] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam and D. Burger, "Dark silicon and the end of multicore scaling", 2011 in 38th Annual International Symposium on Computer Architecture (ISCA)

[2] Thomas N. Theis and H. S. Philip Wong. 2017. "The End of Moore's Law: A New Beginning for Information Technology", 2017 in Computing in Science & Engineering, DOI: https://doi.org/10.1109/MCSE.2017.29

[3] Shekhar Borkar, "Thousand Core Chips: A Technology Perspective", 2007 in Proceedings of the 44th annual Design Automation Conference, DOI: https://doi.org/10.1145/1278480.1278667

[4] David W. Wall, "Limits of instruction-level parallelism", 1991 in Proceedings of the Fourth International Conference on Architectural support for programming languages and operating systems, DOI: https://doi.org/10.1145/106972.106991

[5] Alvaro Estebanez, Diego R. Llanos and Arturo Gonzalez-Escribano, "A Survey on Thread-Level Speculation Techniques.", 2016 in ACM Comput. Surv. 49, 2, Article 22 DOI: https://doi.org/10.1145/2938369

[6] Monica S. Lam and Robert P. Wilson, "Limits of control flow on parallelism.", 1992 in Proceedings of the 19th annual international symposium on Computer architecture, DOI: https://doi.org/10.1145/139669.139702

[7] Manohar K. Prabhu and Kunle Olukotun, "Using thread-level speculation to simplify manual parallelization", 2003 in Proceedings of the ninth symposium on Principles and practice of parallel programming, DOI: https://doi.org/10.1145/781498.781500

[8] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry, "Compiler optimization of scalar value communication between speculative threads", 2002 in Proceedings of the 10th International conference on Architectural support for Programming Languages and operating systems, DOI: https://doi.org/10.1145/605397.605416

[9] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry, "The STAMPede approach to thread-level speculation.", 2005 in ACM Transactions on Computer Systems pp. 253–300, DOI: https://doi.org/10.1145/1082469.1082471

[10] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas, "POSH: a TLS compiler that exploits program structure", 2006 in Proceedings of the eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, DOI: https://doi.org/10.1145/1122971.1122997

[11] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I., "Parallel-stage decoupled software pipelining.", 2008 in Proceedings of the 6th annual IEEE/ACM International Symposium on Code Generation and Optimization, DOI: https://doi.org/10.1145/1356058.1356074

[12] R. Ranjan, P. Marcuello, F. Latorre and A. González, "P-slice based efficient speculative multithreading,", 2009 in International Conference on High Performance Computing, DOI: https://doi.org/10.1109/HIPC.2009.5433216

[13] Carlos Madriles, Pedro Lopez, Josep Maria Codina, Enric Gibert, Fernando Latorre, Alejandro Martinez, Raul Martinez, and Antonio González, "Anaphase: A Fine-Grain Thread Decomposition Scheme for Speculative Multithreading", 2009 in Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, DOI: https://doi.org/10.1109/PACT.2009.27

[14] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks, "HELIX: automatic parallelization of irregular programs for chip multiprocessing." 2012, in Proceedings of the Tenth International Symposium on Code Generation and Optimization, DOI: https://doi.org/10.1145/2259016.2259028

[15] Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks, "HELIX-RC: an architecture-compiler co-design for automatic parallelization of irregular programs", 2014 in Proceeding of the 41st Annual International Symposium on Computer Architecture, DOI:https://doi.org/10.1145/2678373.2665705

[16] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer and D. Sanchez, "Unlocking Ordered Parallelism with the Swarm Architecture", 2016 in IEEE Micro, vol. 36, no. 3, pp. 105-117, DOI: https://doi.org/10.1109/MM.2016.12

[17] Victor A. Ying, Mark C. Jeffrey, and Daniel Sanchez, "T4: compiling sequential code for effective speculative parallelization in hardware", 2020 in Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture, DOI: https://doi.org/10.1109/ISCA45697.2020.00024

[18] P. Marcuello and A. González, "Thread-spawning schemes for speculative multithreading", 2002 in Proceedings Eighth International Symposium on High Performance Computer Architecture, DOI: https://doi.org/10.1109/HPCA.2002.995698

[19] https://llvm.org/devmtg/2009-10/ScalarEvolutionAndLoopOptimization.pdf

[20] Nigel Stephens, Stuart Biles, Matthias Boettcher , Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico and Paul Walker, "The ARM Scalable Vector Extension", 2017 in IEEE Micro, vol. 37, no. 2, pp. 26-39, DOI: https://doi.org/10.1109/MM.2017.35

[21] P. Marcuello, J. Tubella and A. Gonzalez, "Value prediction for speculative multithreaded architectures", 1999 in Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture, DOI: https://doi.org/10.1109/MICRO.1999.809461

[22] Niall Murphy, "Discovering and exploiting parallelism in DOACROSS loops", PhD Thesis, 2016, University of Cambridge, Computer Laboratory, https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-882.pdf

[23] Paraskevas Yiapanis, Gavin Brown, and Mikel Luján. 2015. Compiler-Driven Software Speculation for Thread-Level Parallelism.", 2016 in Transactions on Programming Languages and Systems, 38, 2, Article 5, DOI: https://doi.org/10.1145/2821505

[24] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson, "Using memory mapping to support cactus stacks in work-stealing runtime systems.", 2010 in Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, DOI: https://doi.org/10.1145/1854273.1854324

[25] Leslie Lamport, "The parallel execution of DO loops", 1974 in Commun. ACM 17, 2, 83–93. DOI: https://doi.org/10.1145/360827.360844

[26] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval, " How much parallelism is there in irregular applications?" 2009 in Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, DOI: https://doi.org/10.1145/1504176.1504181

[27] François Irigoin, Pierre Jouvelot, and Rémi Triolet, "Semantical interprocedural parallelization: an overview of the PIPS project", 1991 in Proceedings of the 5th international Conference on Supercomputing, DOI: https://doi.org/10.1145/109025.109086

[28] K. Kennedy, K. S. McKinley, and C. W. Tseng, "Interactive parallel programming using the ParaScope Editor", 1991 in IEEE Transactions on Parallel and Distributed Systems, vol. 2, no. 3, pp. 329-341, DOI: https://doi.org/10.1109/71.86108

[29] T. Brandes, S. Chaumette, M. C. Counilh, J. Roman, A. Darte, F. Desprez, and J. C. Mignot, "HPFIT: a set of integrated tools for the parallelization of applications using High Performance Fortran. PART I: HPFIT and the TransTOOL environment.", 1997 in Parallel Computing, DOI: https://doi.org/10.1016/S0167-8191(96)00097-X

[30] Makoto Ishihara, Hiroki Honda, and Mitsuhisa Sato, "Development and Implementation of an Interactive Parallelization Assistance Tool for OpenMP: iPat/OMP.", 2006 in Transactions on Information and Systems, DOI: https://doi.org/10.1093/ietisy/e89-d.2.399

[31] D. A. Padua, R. Eigenmann, Jay Hoeflinger, Paul Petersen, Peng Tu, Stephen Weatherford, Keith Faigin, "Polaris: A new-generation parallelizing compiler for MPPs.", Technical report, In CSRD No. 1306. UIUC, 1993.

[32] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, Shih-Wei Liao, E. Bugnion and M.S Lam., "Maximizing multiprocessor performance with the SUIF compiler.", 1996 in Computer, vol. 29, no. 12, pp. 84-89, DOI: https://doi.org/10.1109/2.546613

[33] Open64. http://www.open64.net

[34] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall, "The implementation of the Cilk-5 multithreaded language.", 1998 in Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, DOI: https://doi.org/10.1145/277652.277725

[35] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe, "A stream compiler for communication-exposed architectures.", 2002 in ACM SIGOPS Operating Systems Review 36, 5, 291–303, DOI:https://doi.org/10.1145/635508.605428

[36] P. Husbands Parry, C. Iancu, and K. Yelick., "A performance analysis of the Berkeley UPC compiler.", 2003 in Proceedings of

the 17th annual international conference on Supercomputing SC, DOI:https://doi.org/10.1145/782814.782825

[37] Vijay A. Saraswat, Vivek Sarkar, and Christoph von Praun., "X10: concurrent programming for modern architectures.", 2010 in Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, DOI: https://doi.org/10.1145/1229428.1229483

[38] J. Torrellas, "Thread-Level Speculation", 2011, Encyclopedia of Parallel Computing. DOI: https://doi.org/10.1007/978-0-387-09766-4_170

[39] J. Salamanca, J. N. Amaral and G. Araujo, "Using Hardware-Transactional-Memory Support to Implement Thread-Level Speculation", 2018 in IEEE Transactions on Parallel and Distributed Systems, DOI: https://doi.org/10.1109/TPDS.2017.2752169

[40] Yiannakis Sazeides and James E. Smith., "The predictability of data values.", 1997 in Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture.

[41] R. Odaira and T. Nakaike, "Thread-level speculation on off-the-shelf hardware transactional memory,", 2014 in IEEE International Symposium on Workload Characterization, DOI: https://doi.org/10.1109/IISWC.2014.6983060

[42] Maurice Herlihy and J. Eliot B. Moss., "Transactional memory: architectural support for lock-free data structures." 1993 in SIGARCH Computer Architecture News, 289–300. DOI: https://doi.org/10.1145/173682.165164

[43] T. N. Vijaykumar, S. Gopal, J. E. Smith, and G. Sohi, "Speculative versioning cache", 2001 in IEEE Transactions on Parallel and Distributed Systems, vol. 12, no. 12, pp. 1305-1317, doi: https://doi.org/10.1109/71.970565

[44] S. C. Goldstein, K. E. Schauser, and D. E. Culler, "Lazy threads: Implementing a fast parallel call", 1996 in Journal of Parallel and Distributed Computing, DOI: https://doi.org/10.1006/jpdc.1996.0104

[45] Chuan-Qi Zhu and Pen-Chung Yew, "A Scheme to Enforce Data Dependence on Large Multiprocessor Systems" 1987 in IEEE Transactions on Software Engineering, vol. SE-13, no. 6, pp. 726-739, DOI: https://doi.org/10.1109/TSE.1987.233477

[46] G. Ottoni, R. Rangan, A. Stoler and D. I. August, "Automatic thread extraction with decoupled software pipelining", 2005 in 38th Annual IEEE/ACM International Symposium on Microarchitecture, DOI: https://doi.org/10.1109/MICRO.2005.13

[47] Nikolas Ioannou, Jeremy Singer, Salman Khan, Polychronis Xekalakis, Paraskevas Yiapanis, Adam Pocock, Gavin Brown, Mikel Lán and Marcello Cintra, "Toward a more accurate understanding of the limits of the TLS execution paradigm", 2010 in IEEE International Symposium on Workload Characterization, DOI: https://doi.org/10.1109/IISWC.2010.5649169

[48] P. Marcuello and A. González, "A quantitative assessment of thread-level speculation techniques". 2000 in Proceedings of the 14th International Parallel and Distributed Processing Symposium, DOI: https://doi.org/10.1109/IPDPS.2000.846040

[49] J. T. Oplinger, D. L. Heine and M. S. Lam, "In search of speculative thread-level parallelism," 199 in International Conference on Parallel Architectures and Compilation Techniques, DOI: https://doi.org/10.1109/PACT.1999.807576

[50] F. Warg and P. Stenström, "Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms", 2001 in Proceedings of International Conference on Parallel Architectures and Compilation Techniques, DOI: https://doi.org/10.1109/PACT.2001.953302

[51] Arun Kejariwal, Xinmin Tian, Wei Li, Milind Girkar, Sergey Kozhukhov, Hideki Saito, Utpal Banerjee, Alexandru Nicolau, Alexander V. Veidenbaum, and Constantine D. Polychronopoulos, "On the performance potential of different types of speculative thread-level parallelism.", 2006 in Proceedings of the 20th annual international conference on Supercomputing, DOI: https://doi.org/10.1145/1183401.1183407

[52] Arun Kejariwal, Xinmin Tian, Milind Girkar, Wei Li, Sergey Kozhukhov, Utpal Banerjee, Alexander Nicolau, Alexander V. Veidenbaum, and Constantine D. Polychronopoulos "Tight analysis of the performance potential of thread speculation using spec CPU 2006", 2007 in Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, DOI: https://doi.org/10.1145/1229428.1229475

[53] V. Packirisamy, A. Zhai, Wei-Chung Hsu, P. Yew and T. Ngai, "Exploring speculative parallelism in SPEC2006", 2009 in IEEE International Symposium on Performance Analysis of Systems and Software, DOI: https://doi.org/10.1109/ISPASS.2009.4919640

[54] Manohar K. Prabhu and Kunle Olukotun, "Exposing speculative thread parallelism in SPEC2000.", 2005 in Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, DOI: https://doi.org/10.1145/1065944.1065964

[55] Samuel P. Midkiff., "Automatic Parallelization. An overview of fundamental compiler techniques.", 2012, Morgan and Claypool Publishers, DOI: https://doi.org/10.2200/S00340ED1V01Y201201CAC019

[56] Paraskevas Yiapanis, Demian Rosas-Ham, Gavin Brown, Mikel Luján, "Optimizing software runtime systems for speculative parallelization" 2013 in Transactions on Architecture and Code Optimization, DOI: https://doi.org/10.1145/2400682.2400698