

A Novel Integrated Parallel Accelerator for an Irregular Killer App

Yuhao Song
songyh@umd.edu
University of Maryland
College Park, Maryland, USA

Manoj Franklin
manoj@umd.edu
University of Maryland
College Park, Maryland, USA

Uzi Vishkin
vishkin@umd.edu
University of Maryland
College Park, Maryland, USA

Abstract

With severe challenges facing computer technology scaling, hardware accelerators have become increasingly attractive solutions to continue expected generational scaling of performance, as demonstrated by the outstanding commercial success of GPUs for regular parallelism applications such as computer graphics and machine learning. However, parallelism scaling is yet to have a similar impact on irregular algorithms and applications, where distributions of work and data are input-dependent and therefore defy compile-time characterization; e.g., (i) characterizing complex, pointer-based data structures in graph algorithms for locality or load balancing, and (ii) characterizing the transient embedding of heavily nested, input-dependent, fine-grained threads on parallel hardware. Furthermore, when parallelism in such algorithms and applications is limited, overheads in hardware and software exploitation techniques tend to dominate performance, leading even to slow downs, rather than speedups, over standard serial CPUs.

Taking the bull by its horns, this paper advances a powerful idea for drastically speeding up irregular parallelism applications. Our goal is to develop a commercially viable parallel software-hardware framework that is especially geared for speeding up irregular applications. To that end, we select a commercially valuable application (“killer app”), whose irregular parallelism requires overcoming extremely challenging aspects of irregular algorithms, such as: (i) recursive spawning of short threads, and (ii) focusing on the most problematic end where parallelism is limited. The application we have selected is SAT solver, a core routine that does the heavy computational lifting in a vast majority of automated reasoning applications. We first develop parallelism models that involve nested spawning of threads. Then, we introduce NPA (Nested Parallel Accelerator), an integrated hardware accelerator that efficiently implements these nested parallelism models with low overhead. We have limited modifications to current CPU core to a few “hooks” to improve the chances for adoption by industry. Experimental results using a detailed microarchitectural simulator show speedups of up to 17.5X when using 10 execution units (and 50 “reservation stations”). Increasing the number of execution units to 50, the maximum speedup goes up to 36X. Due to its diminishing scaling, our NPA should best be used in conjunction with prior techniques for workloads that have more parallelism.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference acronym 'XX, Woodstock, NY

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/2018/06

<https://doi.org/XXXXXXXX.XXXXXXX>

CCS Concepts

• **Computer systems organization** → **Architectures; Parallel architectures; Multicore architectures;**

Keywords

Irregular parallelism, Join-free nested spawn, Low overhead spawn, Parallel algorithms, SAT solver

ACM Reference Format:

Yuhao Song, Manoj Franklin, and Uzi Vishkin. 2025. A Novel Integrated Parallel Accelerator for an Irregular Killer App. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 13 pages. <https://doi.org/XXXXXXXX.XXXXXXX>

1 Introduction

How to parallelize irregular applications—an important domain of software applications—has long been a leading grand challenge for the computer systems community. Better leveraging the full potential of the ongoing exponential transistor scaling is necessary to bring many more applications back into the historical fold of exponential performance growth as well as significant power savings. Indeed, there have been many proposals to address this grand challenge, whether by using: (i) software-only methods on multicores [11, 34, 47], (ii) software-only methods on domain-specific architectures such as GPUs [40], (iii) dedicated hardware for a specific application [17, 26, 27, 49], or (iv) software techniques in combination with hardware upgrades such as XMT [54, 56]. While the first two approaches achieved major impact on high-value commercial regular applications, extension to irregular applications has not. They have major shortcomings for many irregular applications due to their high communication and synchronization overheads. The use of dedicated hardware requires the design and development of highly specialized large hardware systems that can only do a specific application and no other application.

The fourth approach—using software techniques along with hardware upgrades—has not yet gained sufficient traction from vendors. In our opinion, a case must still be made that high speedups can be achieved for **heavy weight** applications deemed investment worthy (henceforth, “killer apps”). Making such a case would be transformative to the frontier of knowledge, removing the biggest current obstacle towards overcoming the above grand challenge. Such removal requires addressing a “dirty little secret” of irregular parallel computing: overheads. This paper identifies and addresses specific fault lines between parallel algorithms and their current implementation whose overheads stand in the way of achieving compelling speedups for said killer apps.

1.1 SAT Solver — A Killer-App-First Approach

We propose an integrated hardware accelerator that can much better utilize the parallelism present in irregular applications. Towards that end, it is important to focus on a challenging, irregular killer application. Killer applications have the potential to drive the development of high-end computer systems and to attract large investments from the processor industry. Success with challenging applications improves the prospects for success with other perhaps less challenging applications.

The killer app we consider in this paper is SAT solving — an important problem in many domains including formal verification, security analysis, and planning. A SAT solver attempts to find an assignment of Boolean values to variables that result in a given formula being true. It is also possible to reduce any NP problem to SAT in polynomial time [14]. SAT solvers are under continuous development, with an annual SAT competition highlighting the latest advances [5].

Modern SAT solvers spend almost 90% of the time in *unit clause propagation* (aka Boolean Constraint Propagation), a low-level routine which is at the heart of various heuristics employed to reduce the search space [17]. The parallelism present in this routine is *irregular, nested and dynamically varying* [20]. Furthermore, initial truth assignments tend to provide limited parallelism even in cases where its direct and indirect implications provide more parallelism.

1.2 Current Approaches to Parallelizing SAT Solvers

There is a long record of optimizing SAT solvers, resulting in many practical problems being translated into SAT instead of being solved directly [21, 39]. Indeed, SAT solvers leverage much broader forms of automated reasoning in solvers known as SMT, for satisfiable moduli theories, supported by large research and development groups at companies such as Microsoft and Amazon (e.g., [12, 15, 23, 28, 33, 35, 45, 46, 50, 55]), and other fields (e.g., bioinformatics [37] and computer security [42]). Generative AI techniques, such as GPT, recently demonstrated their potential for automatically producing code. As such code cannot be trusted, it has been up to software developers to determine correctness, defeating the very purpose of automation. SAT and SMT solvers can contribute towards automating correct code production. For example, derive both specs and code using GPT, then: (1) use solvers to verify that code correctly meets the specs, or (2) incorporate solvers in a reinforcement-learning-type feedback loop. Related verification challenges involve translating code in one language (e.g., Matlab) to another (e.g., Python), where the specs are clear.

State-of-the-art approaches to parallel SAT solving in a **multi-core system**, such as Glucose-Syrup [1, 2], Painless [22], Plingeling [6], and Treengeling [8], fall within approach (i) above (software-only). They rely on coarse-grain kernel-level threads that run the same CNF formula with different heuristics while sharing useful information on learned clauses, and is known as the *Portfolio approach*. Such approaches map well to current high-end computer systems, with one or more sequential solvers per node. However, they have so far demonstrated only a limited scalability (e.g., [29, 32]), as simple strategies can result in redundant work and complex ones can

encounter high communication overheads or significant sequential computation up front or both.

Speeding up the time-consuming parts of SAT solvers requires parallelizing the *unit propagation* routine, which boils down to supporting **nested spawning** of fine-grain, irregular threads. Some previously proposed software-only approaches for explicit handling of nesting include: [10] and later work on the NESL language, [9] and later work on the Cilk project, and quite a few "lazy" schedulers of coarse-grained nested parallelism including [43]. We parallelized the unit propagation step of a SAT Solver with Cilk and ran the parallel code on a 32-core (64-thread) Intel Xeon Gold system. However, the overheads were high, and we observed no speedup. Manthey was able to get a modest speedup of 7% for unit propagation with the use of spin locks (to avoid system calls) and better load balancing [38].

There have been attempts to parallelize SAT solvers using GPUs (approach (ii) mentioned above) [16, 24, 31, 41, 48]. Some of them show only modest speedups and that too only relative to suboptimal serial code. This is to be expected, as GPUs are specifically designed to target massively regular parallelism.

There have also been proposals to speed up SAT solvers using dedicated hardware [57] that implement in hardware (either ASIC [27, 49] or FPGA [17, 26]) either the entire SAT solver solution [27] or major components of the solution [17, 49], such as ad-hoc data restructuring. For instance, Govindasamy, et al. [26] report up to 6X speedup for unit propagation on an FPGA-based accelerator. The survey [25] reviews several projects utilizing special-purpose FPGAs for instances of SAT solving. While these accelerators generally provide better speedups than the software approaches that utilize multicores or GPUs, they require the design and development of highly specialized large hardware systems that can only do SAT solving and no other application.

Finally, approach (iv)—the use of software techniques along with hardware upgrades—has not yet been tried for SAT solvers. While XMT generally represents this approach, the fine-grained software work in [52] still encountered high overheads for nested spawning of threads.

1.3 Paper Objectives

There has been no major effort to successfully parallelize in software the *unit clause propagation* routine, which has *irregular, nested and dynamically varying* parallelism [20]. The use of multiple threads on commercial hardware platforms encounter penalizing overheads due to: (i) thread initialization overheads and (ii) inefficient handling of fine-grained nested threads. Limited parallelism encounters significant slowdown rather than speedup over best serial counterparts.

Seeking to minimize the overheads for the initiation of short fine-grained nested threads and their operation, our paper aims to replace traditional software nested threading techniques as needed for parallelizing the untapped parallelism present in unit clause propagation by novel hardware/software techniques. This parallelism can be combined with better understood, coarse-grain forms of parallelism, such as Portfolio and Divide-and-Conquer. We seek to provide *speedups on top of what is already achieved by existing coarse-grained parallel approaches*.

Thus, our specific goals are to propose a low-overhead hardware techniques for fine-grained nesting and evaluate its performance using a detailed microarchitectural simulator. We also aspire to make the accelerator general enough to accelerate a variety of other irregular applications such as parallel BFS and parallel DFS.

Our goals in this paper are to: (i) Develop an upgraded computing stack centered around a novel integrated parallel accelerator and comprising algorithms, whose objective is to speed up the unit propagation step in order to exploit the untapped parallelism in SAT solvers (ii) Prototype the proposed hardware with a microarchitectural simulator. This work presented in this paper comprises two inter-related components.

The first component involves a promising approach to parallelize the unit propagation **algorithm** in Glucose [1, 2], a state-of-the-art general-purpose sequential SAT solver, while preserving its carefully developed heuristics. Because Glucose uses the same algorithmic framework as most other modern SAT solvers, we expect that insights gained from simulations with Glucose can be applied to SAT solvers more broadly.

The second component involves our development of an **architecture** to efficiently exploit the additional parallelism unearthed in the first component. In doing so, we have developed hardware techniques to mitigate various performance hurdles. For example, as the additional parallelism we unearthed is fine-grained and irregular, including fine-grained nested parallelism, we have developed hardware techniques to best support these features. This component has been guided by a **microarchitectural simulator** that we developed to faithfully model the hardware components of our design.

1.4 Contributions of this Paper

The major contributions of this paper are as follows:

- (1) Parallelization with nested spawning as well as no “joins”
- (2) Development of a novel low-overhead integrated hardware accelerator, called the Nested Parallel Accelerator (NPA)
- (3) Implementation of the proposed parallelization model with NPA that:
 - (a) Prefetches the irregular memory accesses of spawned threads before activating them so that activated threads do not wait for memory.
 - (b) Performs out-of-order activation of threads based on their readiness to execute.
 - (c) Allows threads to terminate immediately after completion so that they do not wait for their child threads to complete and then perform a “join” operation.

The fundamental principle here is that an activated thread undergoes minimal waiting and occupies the thread execution unit for only a short time.
- (4) Evaluation of NPA with a detailed microarchitectural simulator.
- (5) Demonstration of strong speedups in spite of limited application parallelism, an important and generally under-addressed concern.

The remainder of this paper is organized as follows: Sec. 2 presents our proposed algorithmic techniques to parallelize unit

propagation in SAT solvers. Sec. 3 discusses architectural challenges and presents our NPA. Sec. 4 presents experimental results obtained with a detailed microarchitectural simulator that we have developed. Sec. 5 provides a summary and future work.

2 Parallelizing SAT Solvers

A *Boolean variable* (e.g., x) can take one of two values: *TRUE* or *FALSE*. A *literal* is a variable x or its negation (written $\neg x$). If x is *TRUE*, its negation $\neg x$ is *FALSE* and vice versa. A clause is a disjunction of literals (e.g., $x \vee \neg y \vee z$); a clause is *TRUE* (also called satisfied) if at least one literal in the clause is *TRUE* and *FALSE* (also called unsatisfiable) if all literals are *FALSE*. A unit clause is a clause with exactly one literal. A *formula in conjunctive normal form (CNF)* is a conjunction of clauses (e.g., $(x \vee \neg y) \wedge (y \vee z)$); a CNF formula is *TRUE* (satisfied) if all clauses in the formula are *TRUE* and *FALSE* (unsatisfiable) if at least one clause is *FALSE*. The Boolean satisfiability problem (SAT) is to decide whether a given formula is satisfiable.

2.1 Sequential SAT Solver Algorithms

The Boolean satisfiability problem is NP-complete: no polynomial time for SAT solving exists or is likely to be found in the future. Therefore, efficient SAT solvers rely on heuristics to provide good performance on practical inputs. The most common approach for SAT solving is the backtracking search algorithm. It traverses the search tree of all partial variable assignments in a depth-first manner until it finds a satisfying assignment or until it concludes that no such assignment exists and the formula is unsatisfiable. Most modern SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [18] (see Algorithm 1). It traverses the search tree, depth-first, one variable at a time, assigning it either the *TRUE* or *FALSE* value. The worst-case time complexity of all backtracking algorithms is exponential in the number of variables. To improve performance, modern SAT solvers are engaged in 3 performance enhancing steps, aimed at pruning the search space explored.

- **Unit propagation.** After each assignment, the CNF formula is simplified by unit propagation (see Algorithm 2); a form of constraint propagation that prunes the search space while moving forward, extending a partial solution by one more variable. If unit propagation generates an empty clause, the current assignment is declared a conflict; DPLL backtracks to the last variable it assigned, flips its value (from *TRUE* to *FALSE* or vice versa), and tries again.
- **Variable and value ordering heuristics.** Unit propagation is also instrumental in facilitating look-ahead heuristics for selecting the next variable and its next value. These ordering decisions are known to have an immense impact on the size of the search tree explored, and thus on the running time of the algorithm.
- **Conflict-Directed Clause Learning (CDCL).** When a conflict occurs, rather than naively backtracking to a previous assignment, the algorithm analyzes the reason for the conflict, and learns a new clause (or a *nogood*) that is added to the CNF formula, ensuring that the same conflict will not occur during the remainder of the search. In addition,

Algorithm 1 Conflict Directed Clause Learning (CDCL) Solver

Input: A CNF formula ϕ
Output: A decision of whether ϕ is satisfiable

```

1: function CDCL( $\phi$ )
2:   if not UNIT_PROPAGATE( $\phi$ ) then
3:     return UNSAT
4:   end if
5:   while  $\phi$  has an unassigned variable do
6:      $(x, v) \leftarrow$  PICKBRANCHVAR( $\phi$ )  $\triangleright$  Decide var  $x$  and value  $v$ 
7:      $\phi \leftarrow \phi \wedge (x = v)$   $\triangleright$  Substitute  $x = v$  in  $\phi$ 
8:     if not UNIT_PROPAGATE( $\phi$ ) then  $\triangleright$  Deduce stage
9:        $\beta =$  CONFLICTANALYSIS( $\phi$ )  $\triangleright$  Diagnose stage
10:      if  $\beta < 0$  then
11:        return UNSAT
12:      else
13:        BACKJUMP( $\phi, \beta$ )
14:      end if
15:    end if
16:  end while
17:  return SAT  $\triangleright$  return satisfiability
18: end function

```

Algorithm 2 Unit Propagation (Serial Algorithm)

Output: An equivalent formula with no unit clauses

```

1: function UNIT_PROPAGATE( $\phi$ )
2:    $Queue \leftarrow$  all unit clause literals in  $\phi$ 
3:   while  $Queue$  is not empty do  $\triangleright$  "Outer" loop
4:      $U \leftarrow$  next literal from  $Queue$ 
5:     for each clause  $C$  containing  $U$  or  $\neg U$  do  $\triangleright$  "Inner" loop
6:       if  $C$  contains  $U$  then
7:         Delete  $C$  from  $\phi$   $\triangleright$  subsumption elimination [19]
8:       else
9:         Delete  $\neg U$  from  $C$   $\triangleright$  Known as resolution [19]
10:        if  $C$  is empty then return FALSE  $\triangleright$  Conflict
11:        if  $C$  is a unit clause
12:          then add  $C$ 's variable to  $Queue$ 
13:        end if
14:      end for
15:    end while
16:  return TRUE
17: end function

```

instead of backtracking just one level, the algorithm can *backjump* multiple levels, thereby shortcutting parts of the search space that cannot lead to a solution.

2.2 Proposed Parallelization of Unit Propagation

Modern SAT solvers spend about 90% of the time in the unit propagation step (Algorithm 2), which has high amounts of parallelism [20]. However, parallelization of this important unit propagation step has been challenging, as this parallelism is irregular and the amount of parallelism keeps varying dynamically. This paper focuses on augmenting the traditional portfolio parallelization efforts

with parallelization of this crucial unit propagation step, to provide significant speedup on top of what is already possible.

As written, the unit propagation step is a doubly nested loop akin to a breadth-first search (BFS) of a bipartite graph whose nodes are variables and clauses, with an edge connecting each clause to the variables it contains, as shown in Figure 1. Because the BFS frontier consists of unit clauses, which have a fanout of 1, we can compress the traversal from unit clauses to variables and back to clauses into a single unit propagation step; this "compressed BFS" will henceforth be simply called "BFS". A simple-minded approach would be to parallelize just the inner loop, with no nested spawning.

Although parallelization of the inner loop alone may provide some additional speedup, more speedup is possible if we flatten the nested loop into a two-loop sequence and parallelize the flattened loops, as given in Algorithm 3. The UNIT_PROPAGATE() function of Algorithm 3 thus has two **spawn-joins**, one after the other. The threads produced by the first **spawn** execute the function PROPAGATE() to identify in parallel all of the clauses containing each of the unit clause literals. After performing a join, the parent thread performs a **spawn** to produce threads execute the function ELIMINATE_RESOLVE() to process all of the identified clauses in parallel. The ELIMINATE_RESOLVE() threads may add additional unit clause literals to the *Literals* set, which will trigger additional **spawns**, but only after all of the threads spawned in Line 6 of Algorithm 3 have joined. One single thread can delay this join operation thereby delaying the next round of spawning.

Algorithm 3 Parallel Unit Propagation with Spawn-Join

Output: An equivalent formula with no unit clauses

```

1: shared  $\phi, rv = TRUE$ 
2: function UNIT_PROPAGATE
3:    $Literals \leftarrow$  all unit clause literals in  $\phi$ 
4:   spawn PROPAGATE( $U$ ) for each  $U$  in  $Literals$   $\triangleright$  "Outer"
5:   join
6:   spawn ELIMINATE_RESOLVE( $C, U$ ) for each  $C$  in  $Clauses$ 
7:   join
8:   if  $Literals$  is not empty then goto line 3
9:   return  $rv$ 
10: end function
11: function PROPAGATE( $U$ )
12:    $Clauses \leftarrow$  all clauses in  $\phi$  containing literal  $U$  or  $\neg U$ 
13: end function
14: function ELIMINATE_RESOLVE( $C, U$ )
15:   if  $C$  contains  $U$  then
16:     delete  $C$  from  $\phi$   $\triangleright$  subsumption elimination [19]
17:   else
18:     acquire  $C$ 's lock
19:     delete  $\neg U$  from  $C$   $\triangleright$  Known as resolution [19]
20:     release  $C$ 's lock
21:     if  $C$  is empty then  $rv = FALSE$   $\triangleright$  Conflict
22:     else if  $C$  is a unit clause then add  $C$ 's variable to  $Literals$ 
23:   end if
24: end function

```

A lot more potential speedup is possible when both the inner and outer loops of Algorithm 2 are parallelized [20] in a nested

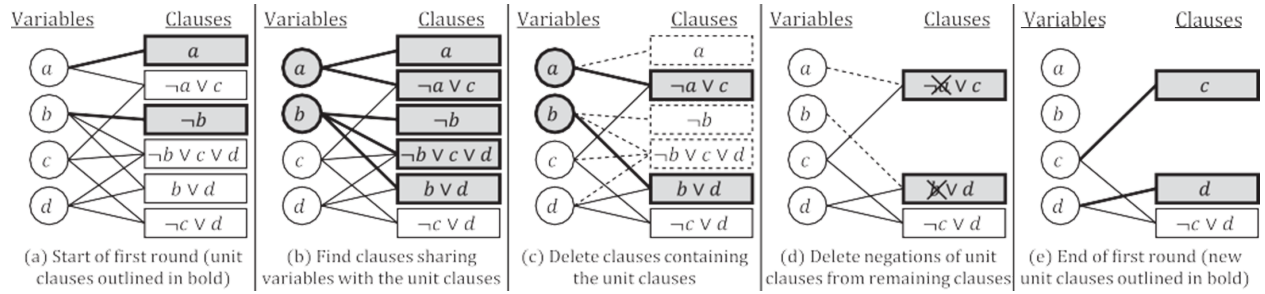


Figure 1: Unit propagation viewed as parallel graph traversal

spawn-join manner, as depicted in Fig. 2(a). In the figure, the parent thread, shown in black, spawns 4 child threads (shown in blue). Among these 4 child threads, the last one spawns 3 child threads of its own (shown in red) in a nested manner. All of the sibling threads need to complete a *Join* operation before their parent can continue. Algorithm 4 shows our nested spawn-join parallel version of Algorithm 2. We parallelize the outer loop by processing multiple unit clauses in parallel and the inner loop by processing all clauses containing a unit clause variable in parallel. Our algorithm uses nested spawning by allowing a child thread to spawn a new thread for each new unit clause variable as soon as it is discovered to be a unit clause. This allows threads to start as early as possible (if we develop a hardware platform that supports efficient dynamic spawning of new threads within parallel code).

Algorithm 4 Parallel Unit Propagation with Nested Spawn-Join

```

1: shared  $\phi$ ,  $rv = TRUE$ 
2: function UNIT_PROPAGATE()
3:    $Literals \leftarrow$  all unit clause literals in  $\phi$ 
4:   spawn PROPAGATE( $U$ ) for each literal  $U$  in  $Literals$ 
5:   join ▷ Wait for all of its child threads
6:   return  $rv$ 
7: end function
8: function PROPAGATE( $U$ )
9:    $Clauses \leftarrow$  all clauses in  $\phi$  containing literal  $U$  or  $\neg U$ 
10:  spawn ELIMINATE_RESOLVE( $C, U$ ) for each  $C$  in  $Clauses$ 
11:  ▷ Nested spawning
12:  join ▷ Wait for all of its child threads
13: end function
14: function ELIMINATE_RESOLVE( $C, U$ )
15:  if  $C$  contains  $U$  then
16:     $\phi \leftarrow \phi \setminus C$  ▷ Clause  $C$  is satisfied; delete  $C$  from  $\phi$ 
17:  else
18:    acquire  $C$ 's lock
19:     $C \leftarrow C \setminus \neg U$  ▷ Delete  $\neg U$  from clause  $C$ 
20:    release  $C$ 's lock
21:    if  $C$  is empty then  $rv = FALSE$  ▷ Conflict detected
22:    else if  $C$  is unit clause then ▷ Let  $C$ 's literal be  $V$ 
23:      PROPAGATE( $V$ ) ▷ Will cause recursive spawning
24:    end if
25:  end if
26: end function

```

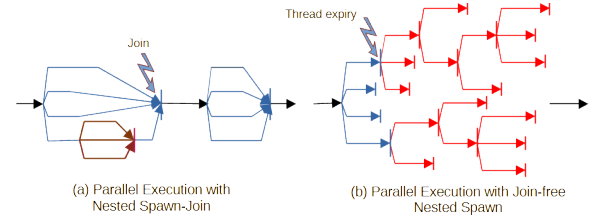


Figure 2: Two Different Nested Spawn Parallelization Models

When a thread completes, its parent performs a join operation. The parent can proceed only after all of its spawned threads have completed. The disadvantage of the spawn-join approach is that the longest thread in a cohort must finish before the cohort's parent thread can continue. The problem is exacerbated when nested spawning is allowed. Moreover, nested spawning raises the hard question of how to implement **nested spawn-joins** with low overheads [51]. Prior nesting-driven work (e.g., [52]) did not seek to optimize overhead for short and repeated spawning. A fundamental question in this case is how to efficiently synchronize a parent thread with its children and grandchildren.

2.3 Proposed Parallelization with Join-Free Nested Spawning

Our technique for efficiently implementing nested spawns is to use a **join-free** approach, as depicted in Fig. 2(b); i.e., a parent does not perform a join operation when each of its child threads completes. More importantly, it does not even keep track of its child threads! Algorithm 5 depicts the unit propagation routine that we have parallelized in this manner. Notice that none of the **spawn** statements in this algorithm have a matching **join** operation. A thread simply expires when its job is finished and does not wait for its child threads (if any). It is important to point out that there is a single **wait** operation in the UNIT_PROPAGATE() function, which waits until all of the created threads have expired, to return a verdict to the CDCL solver running on the CPU core.

Our parallelization technique follows a variant of a SPMD (Single Program Multiple Data) parallelization model; all of the threads generated from a **Spawn** command execute the same set of instructions (which we call a **stencil**), but on different data elements.

A parallel algorithms perspective. The main missing element in prior work is a satisfactory solution for the extremely fine level

Algorithm 5 Parallel Unit Propagation with Join-Free Nested Spawn

```

1: shared  $\phi$ ,  $rv = TRUE$ 
2: function UNIT_PROPAGATE()
3:    $Literals \leftarrow$  all unit clause literals in  $\phi$ 
4:   spawn PROPAGATE( $U$ ) for each literal  $U$  in  $Literals$ 
5:   wait until no threads exist
6:   return  $rv$ 
7: end function
8: function PROPAGATE( $U$ )
9:    $Clauses \leftarrow$  all clauses in  $\phi$  containing literal  $U$  or  $\neg U$ 
10:  spawn ELIMINATE_RESOLVE( $C, U$ ) for each  $C$  in  $Clauses$ 
       $\triangleright$  Nested spawning
11: end function
12: function ELIMINATE_RESOLVE( $C, U$ )
13:  if  $C$  contains  $U$  then
14:     $\phi \leftarrow \phi \setminus C$   $\triangleright$  Clause  $C$  is satisfied; delete  $C$  from  $\phi$ 
15:  else  $\triangleright C$  contains  $\neg U$ 
16:    acquire  $C$ 's lock
17:     $C \leftarrow C \setminus \neg U$   $\triangleright$  Delete  $\neg U$  from clause  $C$ 
18:    release  $C$ 's lock
19:    if  $C$  is empty then  $\triangleright$  Conflict detected
20:       $rv = FALSE$ 
21:      terminate all threads
22:    else if  $C$  is unit clause then  $\triangleright$  Let  $C$ 's literal be  $V$ 
23:      spawn PROPAGATE( $V$ )  $\triangleright$  Recursive spawning
24:    end if
25:  end if
26: end function

```

of nesting granularity and recursion that parallelization of SAT solvers mandates. A secondary though significant issue is that, unlike some other works, we envision SAT solving as a potential killer application for spearheading more general forms of scalable parallelism, ideally for a general-purpose platform.

3 An Integrated Hardware Accelerator for Join-Free Nested Threading

In Section 2, we presented our novel parallelization model—nested spawning of threads with no joins—geared to achieve high speedups for SAT solvers as well as other similar applications. In this section, we propose an integrated hardware accelerator, the Nested Parallel Accelerator (NPA), which efficiently implements the proposed parallelization model.

3.1 Hardware Implementation Challenges

There are many challenges to overcome for efficiently implementing asynchronous spawning in a practical hardware system:

Thread granularity: Our threads are short (typically less than 100 instructions). This makes it unprofitable to run them in current multicores, with kernel-level threads (KLTs), as KLT creation involves a system call and a fair bit of setup. The OS solution typically adopted to deal with this overhead—a *thread pool* (a group of pre-instantiated, idle threads of *similar function*, that stand ready

to be given work)—is not useful for irregular applications whose number of concurrent threads keeps changing significantly. If we create a big thread pool, an excessive number of threads in reserve will waste memory. If we create too small a thread pool, we may not get the benefits of using the thread pool. Irregular applications in fact need a much more adaptive hardware spawn mechanism and a hardware-based thread scheduler that can directly schedule threads onto execution cores.

Thread irregularity: In our proposed parallel algorithm, the length of an ELIMINATE_RESOLVE() thread depends on the clause length and satisfiability. Because of this irregularity, parallel platforms such as GPUs cannot be used effectively. GPUs need regular threads that can be executed in lockstep, incurring performance degradation whenever thread behavior diverges.

Thread spawn overhead: When executing a spawn command in hardware, the hardware spawn mechanism may encounter significant overheads for spawning all of the specified threads. Reasons include: (i) Startup delay for spawning a thread. (ii) Limited system resources. (iii) The bookkeeping overhead for keeping too many threads active.

Thread actuation and scheduling: Even when a thread is spawned, it does not mean that there is enough hardware capacity to execute it. When we have limited hardware resources, we need a mechanism that: (i) dynamically controls which among the active threads is executed, (ii) schedules the execution of spawned threads based on when they are likely to begin execution, and (iii) accounts for the irregularity of memory access and their high latency risks.

Thread communication and synchronization overhead: The communication overheads of fine-grained threads may preclude executing them on multiple CPU cores. However, single-core multithreading techniques such as simultaneous multithreading do not provide scalable speedups.

Irregularity of memory access: The activation of clauses depends on the variables being propagated and their assignment. This means that memory access patterns can only be determined at run time.

Cost-effective handling of the above challenges exceeds the scope of current multi-core and GPU platforms. Efficient implementation of our parallelization model is even more demanding, as it requires nesting of threads, especially with asynchronous (re-)spawning. The start-up cost of new nested threads as well as their allocation to hardware components require rethinking anew the design of the hardware components as well as the allocation management of the code to these components. The hardware that we propose below manages the threads, including their scheduling on available execution units and the required synchronization. We also add low-overhead hardware primitives to support nesting and asynchronous recursive spawning.

3.2 Proposed Hardware Architecture

The basic idea is to augment each main processor core with a parallel processing accelerator/coprocessor that spawns threads asynchronously, schedules them for execution only when they are ready, and lets them expire as soon as they have completed, without having them wait for a “join” operation. Figure 3 gives a block

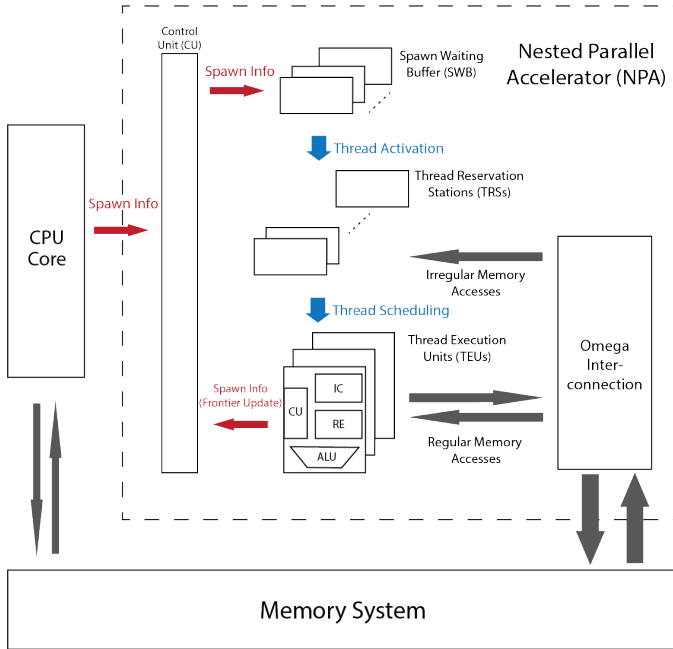


Figure 3: Block Diagram of Proposed Nested Parallel Accelerator (NPA)

diagram of our proposed hardware. The main blocks are a general-purpose CPU core, a Control Unit (CU), a Spawn Waiting Buffer (SWB), Thread Reservation Stations (TRSs), Thread Execution Units (TEUs), an Omega interconnect, and a shared Memory System. Considering that our NPA is a latency-sensitive accelerator, we have made deliberate efforts to minimize the latency between hardware components as much as possible.

• **Spawn Waiting Buffer (SWB)**

The Spawn Waiting Buffer (SWB) maintains information on each pending Spawn command, such as the number of threads and the common starting address of its threads.

• **Thread Reservation Stations (TRSs)**

Thread Reservation Stations (TRSs) serve as lightweight loading units designed to load irregular and indirect memory accesses that are typically present in the beginning part of a thread. The TRS implements only an extremely small subset of RISC-V instructions, specifically lui, add, and lw. This limited instruction set is sufficient for executing short, straightline code segments. (For the threads spawned from the unit propagation routine, this comes to about 10 instructions.) To reduce communication latencies, we organize the TRSs into clusters. The SWB uses a prefix-sum unit [53] to compute dispatch indices in one cycle, enabling fast thread assignment. The SWB buffers are connected to the TRS clusters through an interconnect that supports pipelining and buffering, such as a Mesh-of-Trees interconnect [3]. We are experimenting with different interconnect options to identify the best candidate for this interconnect.

• **Thread Execution Units (TEUs)**

Thread Execution Units (TEUs) are responsible for completing the execution of threads after data preparation in the TRSs. Each TEU is connected to a TRS cluster as shown in Fig. 4. TEUs implement the complete RISC-V instruction set. Most instructions execute in a single clock cycle; however, memory-related operations such as lw and atomic instruction like test-and-set have higher latency depending on cache access times. By offloading early-stage computation and data preparation to light-weight TRSs, the architecture allows TEUs to focus on low-latency operations, thereby enhancing overall parallel efficiency and reducing hardware overhead.

• **Control Unit (CU)**

The Control Unit (CU) serves as the central controller that coordinates the main CPU and the NPA’s internal components, including the SWB, TRSs, and TEUs. The CU maintains status information for all TEUs and TRSs, monitors resource availability, and ensures synchronization. It receives spawn commands issued by the CPU and the TEUs, and forwards them to the SWB. It also handles conflict signals and can halt all threads, giving control back to the main CPU.

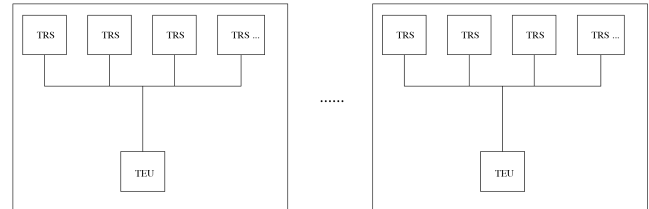


Figure 4: TEU to TRS connectivity.

3.3 ISA Extensions for the Nested Parallel Accelerator (NPA)

To support the NPA and maximize its efficiency, we introduce a minimal set of ISA extensions that enable thread control while avoiding excessive hardware complexity. These instructions are summarized in Table 1, which outlines the supported instructions along with the specific hardware units responsible for their execution.

Table 1: Supported ISA Instructions and Their Associated Hardware Units

Instruction	Supported Hardware Unit(s)
spawn R_{seed} , R_{count} , label	CPU, TEU
trs_halt	TRS
teu_halt	TEU

The semantics and purpose of the new instructions are described below.

- (1) spawn R_{seed} , R_{count} , label:

The spawn instruction is used to initiate a group of threads. It takes the following three arguments:

- R_{seed} : an identifier used to distinguish between different thread groups. In the context of SAT solving, this corresponds to the *literal* of the unit clause group that will be processed.
- R_{count} : the number of threads to spawn within the group. Each thread will receive a unique ID in the range $[0, R_{count} - 1]$.
- *label*: the entry point from which each thread begins execution in TRS.

(2) *trs_halt*:

The *trs_halt* instruction indicates the end of the initialization phase of the thread. Recall that a thread has two parts: a short initialization part, followed by a longer body. The first part executes in a TRS and the second part executes in the TEU connected to the TRS.

(3) *teu_halt*:

The *teu_halt* instruction indicates the end of the thread.

3.4 Working of the NPA

We shall next describe the working of the NPA in detail.

3.4.1 Spawn Instruction Processing. Serial code will execute on the CPU core as usual. When parallel code is reached, as evidenced by a *spawn* instruction (e.g., Line 4 of Algorithm 5), the NPA is activated, which will remain active until all threads have been created and terminated. While the NPA is active, its Control Unit (CU) manages its activities. The CU maintains the information on the pending spawn commands in the Spawn Waiting Buffer (SWB). Every clock cycle, it creates as many threads as the number of available of empty Thread Reservation Stations (TRSs).

3.4.2 Thread Activation and Prefix-Sum-Based Dispatch. In order to dispatch multiple threads in a clock cycle to the available TRSs, a prefix-sum unit [53] is used. The state bit of each TRS—indicating whether it is *Running* or *Idle*—is visible to the SWB. The SWB computes the prefix sum over these state bits in one clock cycle, as illustrated in Figure 5. With this prefix sum information, the SWB can decide in parallel which thread each of the idle TRS will get. Dispatching that thread to the TRS involves only conveying R_{seed} , a unique ID within range $[0, R_{count} - 1]$, and *label*.

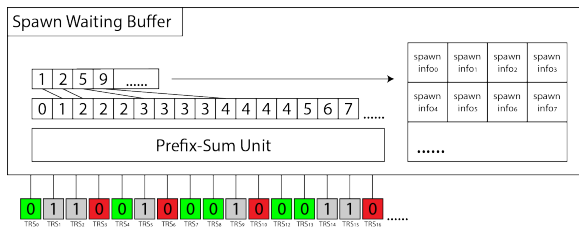


Figure 5: Prefix-sum operation to dispatch threads from the SWB to the TRSs

3.4.3 Thread Initialization Phase in TRS. Upon receiving an allocated thread, the TRS performs the initial memory access(es), which tend to be indirect memory accesses that are likely to suffer higher cache memory miss rates (for the unit propagation case, these would

be the memory references required to fetch the clause needed by that thread). The dispatched thread thus waits in a TRS until it receives its "context" (similar in spirit to an instruction waiting in a reservation station in a dynamically scheduled ILP processor [30]). When the initial memory accesses complete, the TRS executes a *trs_halt* instruction to signal its readiness for further execution in TEU. At this point, the thread is scheduled to an empty Thread Execution Unit (TEU). Notice that the TRS hardware acts like a load unit and as such is a small piece of hardware.

3.4.4 Thread Scheduling and Execution. When a TRS completes execution and the associated TEU is idle, the thread data is transferred to the TEU for execution. The TEU executes the assigned thread using its simple, in-order pipeline, with minimum synchronization with other threads. At the end, there is typically one or more atomic updates to shared memory, followed by the *teu_halt* instruction. Execution of the *teu_halt* instruction causes the TEU to become idle again, signaling its availability for executing another thread. The TEUs maintain a copy of the different threads (*PROPAGATE()* and *ELIMINATE_RESOLVE()* for the unit propagation case) to avoid fetching the executed thread's instructions from the cache memory.

3.4.5 Nested Spawning. While executing a thread, the TEU may encounter the *spawn* instruction (for the unit propagation case, this happens when a clause becomes a unit clause (step 22 of Algorithm 5)), in which case, it initiates a nested spawn operation. When executing a *spawn* instruction, the TEU sends the necessary thread parameters to the CU.

3.5 Salient Features of the Nested Parallel Accelerator (NPA)

Below we highlight the salient features of our proposed NPA.

3.5.1 Hardware-Based Nested Thread Spawning. Threads are created with a *spawn* instruction, without involving the operating system. The *spawn* instruction itself has negligible overhead; it is only telling the NPA that this is a point to create threads. To that end, the *spawn* instruction specifies the number of threads to spawn and any additional information needed to identify the thread's code. Our proposed NPA seamlessly incorporates nested spawning of threads without added overheads. It is able to do so, because our parallelization model does not enforce a strict parent-child relationship between the spawning thread and the spawned threads. This is borne by the fact that the parent does not use "joins" to merge "sibling" threads. Instead, when a thread initiates a (recursive) *spawn*, it vacates the TEU and the TEU can then be repopulated by another ready thread.

3.5.2 Hardware-Based Thread Management. The management of the threads, before and after spawning, is done entirely in hardware, by the Coprocessor Control Unit. It does this management by performing the following functions:

- (1) It creates a "master copy" for spawning threads and places it in the Spawn Waiting Buffer (SWB); when the number of pending *Spawn* commands exceeds capacity, an alternative mechanism kicks in to temporarily suspend further spawning.

- (2) When a Thread Reservation Station (TRS) becomes available, it allocates a newly created thread to the TRS, which then initiates the irregular memory accesses needed to begin the execution of that thread. When these memory values become available, the thread is ready to execute.
- (3) It dispatches a ready thread from a TRS to a Thread Execution Unit (TEU).
- (4) It reclaims a TEU when the allocated thread has executed its last instruction.

3.5.3 Token-Based Thread Allocation. An important innovation of our proposed hardware scheme is that the binding of a spawned thread to a TEU is done only when the thread is ready to execute without significant further delays. Specifically, any memory values a thread needs that stem from irregular memory accesses have been already fetched by the TRS before the thread is assigned to a TEU. Until this binding, the spawned thread waits in a TRS, similar in spirit to an instruction waiting in a reservation station (or load buffer) in Tomasulo’s algorithm [30]. In that sense, our NPA is applying Tomasulo’s dynamic scheduling algorithm at the level of threads. It is important to note that the active threads are executing at their own pace and are not operating in lockstep.

3.5.4 Limited Inter-Thread Synchronization. During the bulk of its execution, a thread only *reads* shared data (clauses) and does not write to shared data. It may update shared data only when it is about to expire. Thus, the threads do not wait for any communication from other threads and can therefore execute at full steam as permitted by memory access latencies. If multiple threads expire at the same time and try to simultaneously update the same shared data, these updates are serialized by means of hardware locks such as test-and-set instructions.

3.5.5 Shared Memory System and Interconnect. In most accelerator designs, the host memory is separate from the accelerator memory (e.g., [13]), and explicit data transfer to and from the accelerator memory is needed to perform the computation and get the results. In irregular applications, the cost of this data transfer is difficult to amortize, since the data reuse in the accelerator is lower than in regular applications. A hallmark of our proposed NPA is that it avoids expensive data transfers between the CPU and the NPA by tightly coupling them—allowing the NPA to utilize the CPU memory system and its address translation mechanism.

The memory system utilizes a hierarchy in order to satisfy low latency. At the top of the hierarchy, we have a shared multi-bank cache memory. The CPU, the TRSs, and the TEUs are connected to the shared cache memory banks through an on-chip Omega interconnect [30]. Misses from the cache memory are routed to the multi-bank main memory.

3.5.6 Power Consumption. The proposed hardware implementation is fundamentally a drastically trimmed version of a much demanding design in terms of both power consumption and silicon area. The NPA does not use speculative execution or extra memory movements, and hence does not increase the energy consumption for solving a specific SAT solver instance.

4 Experimental Results and Analysis

In order to test the efficacy of our proposed hardware accelerator, we perform a detailed quantitative evaluation. For comparison, we evaluate all 3 parallelization models we discussed — Spawn-Join, Nested Spawn-Join, and our proposed Join-Free Nested Spawn — along with the serial model. We also vary the number of thread execution units and thread reservation stations to get more insights on the performance of our NPA.

4.1 Simulator Framework

To do the evaluation, we extended a widely used cycle-accurate microarchitectural simulator based on the RISC-V ISA [36]. Our simulator faithfully models the general-purpose CPU core and the NPA, including the Control Unit (CU), the Spawn Waiting Buffer (SWB), the Thread Reservation Stations (TRSs), the Thread Execution Units (TEUs), the multi-banked cache memory, the Omega network connecting the TRSs and TEUs to the cache banks, and the main memory. The simulator models a conservative 5-cycle latency (based on [3, 4]) for data transfer from the SWB to the TRS clusters; we intend to experiment with different interconnects and reduce this latency. The simulator also models a single-level 1 MB shared cache (5 cycle access time, 8-way set associative, 8 banks) and a 100-cycle latency 8-bank main memory system. When multiple memory accesses map to the same cache bank in the same cycle, only one of them is allowed to proceed; the rest are queued on a per bank basis.

For the Spawn-Join parallelization models (both non-nested and nested), a parent thread will temporarily relinquish its TEU after issuing a *Spawn* command so that its TEU can be used by another thread. Completed child threads will free up their TEUs, but their parent thread is allowed to continue only after all of its child threads have completed (*Join* operation). Our simulator models a hardware *Join Unit* that keeps track of the number of threads that are still active for each *Spawn* command, and the *Spawning Depth* of each thread. For the non-nested Spawn-Join model, we allow all threads that complete in a clock cycle to join in parallel using a prefix-sum unit [53]. The nested Spawn-Join model is challenging to implement on fixed hardware, since the nesting structure can vary greatly from one run to the next. Effective support of the Spawn-Join model requires additional hardware dedicated to thread management. Even when provided, controlling such hardware encounters overheads that penalize the performance of SAT Solver itself. Still, the more concurrent Joins are enabled in the simulation the more the performance gap between the Spawn-Join and Join-Free models narrows.

We incorporated three parallelization models in the Unit Propagation step of Glucose [1, 2], a widely used state-of-the-art SAT solver program. The unit propagation routine in Glucose is **state-of-the-art**. Glucose uses two-watched literal method to optimize SAT solvers where each clause only tracks two of its literals, allowing the solver to skip re-examining a clause unless one of these watched literals becomes false. This strategy minimizes unnecessary checks during Boolean Constraint Propagation, thereby speeding up the SAT Solving [44]. For inputs to Glucose, we use 12 benchmarks from the 2018 SAT Solver Competition. These benchmarks belong to 3 different application domains. Characteristics of these input benchmarks, such as the application domain, number of variables,

Table 2: Benchmark Characteristics

Family	Benchmark	#Variables	#Clauses	Cache miss for TEU
Mutilated-chessboard	2c3	544	1815	0.8%
	457	760	2556	1.0%
	822	612	2048	0.8%
	ec8	420	1391	0.9%
Coloring	17e	19585	140691	5.4%
	28b	21585	158431	4.4%
	3e5	23585	178091	2.9%
	330	25585	196491	4.2%
Cellular-automata	629	42271	248471	2.1%
	6a5	42271	248471	1.7%
	83c	42271	248471	3.1%
	bc7	42271	248471	2.3%

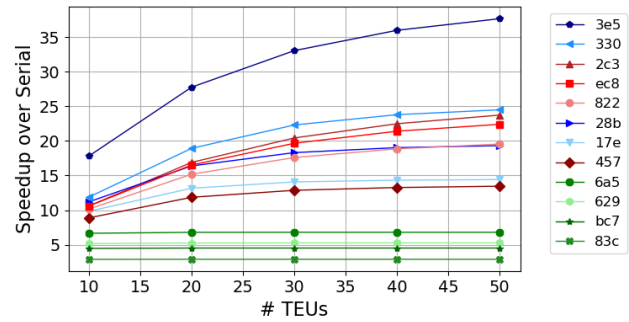
number of clauses, and cache miss ratio for data memory accesses generated from the TEUs (which we observed to be more or less constant) are given in Table 2.

Our experiments with Glucose showed limited parallelism within the unit propagation steps of the SAT solver’s initialization and termination phases. Since these phases account for less than 0.5% of the total unit propagation steps, we excluded the first one million steps to ensure that our simulation samples are representative of the overall computation. The other steps demonstrated similar behavior when it comes to parallelism.

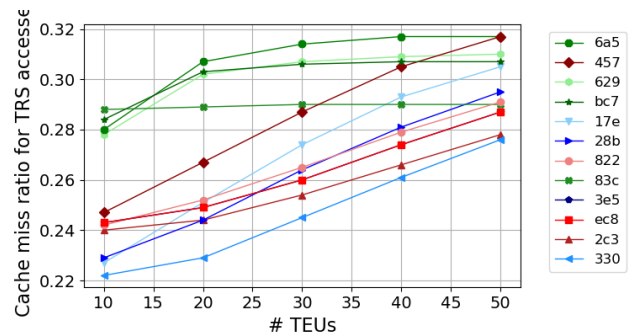
We run simulations for 10,000 consecutive unit propagation steps initiated from the CPU. All reported speedup values reflect the overall speedup over the aggregate runtime of these 10,000 unit propagation steps, compared against an equivalent serial algorithm running on **GEM5**, a state-of-the-art RISC-V simulator [36]; the serial algorithm runs entirely on the simulated high-speed state-of-the-art CPU core and does not utilize the integrated NPA. The CPU configuration simulates a detailed out-of-order core with dynamic scheduling, speculative execution, and a reorder buffer size set to 192.

4.2 Speedup Over Serial Execution

Figure 4 shows the speedups we obtained for our Join-Free Nested Spawn parallelization model (Algorithm 5) over the serial algorithm (Algorithm 2). The X-axis plots the number of TEUs and the Y-axis plots the speedup over the serial algorithm. For these experiments, we kept the number of TRSs as 5 times the number of TEUs. The results show that 8 of the 12 SAT inputs have an impressive speedup of approximately 10 with 10 TEUs, demonstrating the strong potential of our Join-Free Nested Spawning and integrated NPA for speeding up SAT solvers. By providing more reservation units than execution units, we are sometimes able to get speedups that even exceed the number of TEUs. Notice that this speedup is orthogonal to the speedup obtained through previous Portfolio approaches. What enables this accomplishment is our vertically integrated approach: (i) *codesign*, (ii) a novel parallel programming model, and (iii) a novel hardware design. As the number of TEUs is increased

**Figure 6: Speedup of Join-Free Nested Spawn Parallel Code over Serial Code**

to 50 in steps of 10, Figure 6 shows speedups increase for 8 of the 12 benchmarks, although the speedup improvement is diminishing.

**Figure 7: Data Cache Miss Ratios for memory accesses issued by the TRSs; Y-axis starts at 0.22**

In order to get more insight into the speedup numbers, we also report the data cache miss ratios obtained for memory accesses issued from the TRSs (cf. Figure 7). It is interesting to compare these miss ratios with the miss ratios obtained for memory accesses issued from the TEUs (cf. Table 2). We can see that the average miss ratio for TRS memory accesses (0.26 for 10 TEUs) is 10 times that of those issued from the TEUs (averaging 0.025). This is because memory accesses from the TRSs involve indirect (pointer-based) accesses. By including many more TRSs than TEUs, we are able to overlap the processing of cache misses generated by the TRSs, tolerating better their latencies. Another point to note is that those miss rates increase slightly as the number of TEUs (and hence the TRSs) is increased: as the number of TRSs increases, more threads are executed in parallel, resulting in more conflict misses.

4.3 Sensitivity Studies with Number of TRSs

We also performed a sensitivity study by varying the number of Thread Reservation Stations (TRSs) from 10 to 70 (in steps of 10), keeping the number of TEUs fixed at 10. These results, reported in Figure 8, indicate that for those benchmarks with scalable parallelism, it is beneficial to have a higher TRS:TEU ratio; the benefit

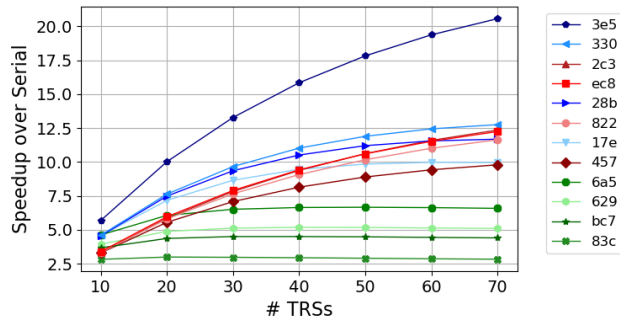


Figure 8: Speedup of Join-Free Nested Spawn Parallelization Model for varying number of TRSs; Number of TEUs is fixed at 10.

starts diminishing around a ratio of 5:1, suggesting that a 5:1 ratio might be an effective empirical setting for the NPA design.

The speedup plots for 2c3, 457, and 822 start at slightly lower values than their cohort but increase more rapidly than theirs. These benchmarks are also the ones that have more scalable parallelism, as evident from the speedup numbers given in Figure 8. The reason for this behavior is twofold: (1) Due to the two-watched literal optimization in Glucose [44], the threads generated from these benchmarks generally run for shorter durations, so the ratio of the overhead of thread creation relative to “useful work” is relatively high. When the number of TRSs is low, the TEUs end up being starved, leading to lower speedups. (2) On the other hand, when the number of TRSs is increased, the overhead of thread creation is amortized by the large number of threads that are spawned due to the higher amounts of parallelism present in these benchmarks.

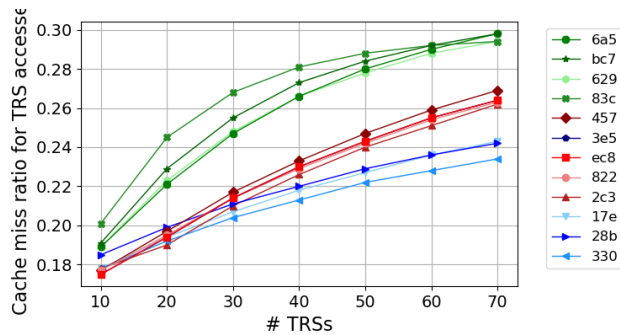
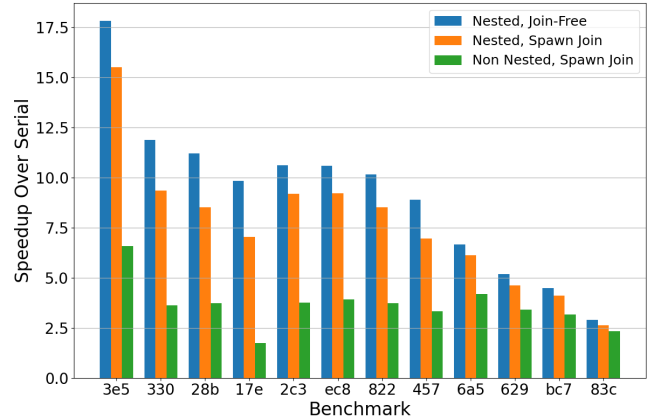


Figure 9: Data Cache Miss Ratios for memory accesses issued by the TRSs, for varying number of TRSs; Number of TEUs is fixed at 10. Y-axis starts at 0.17.

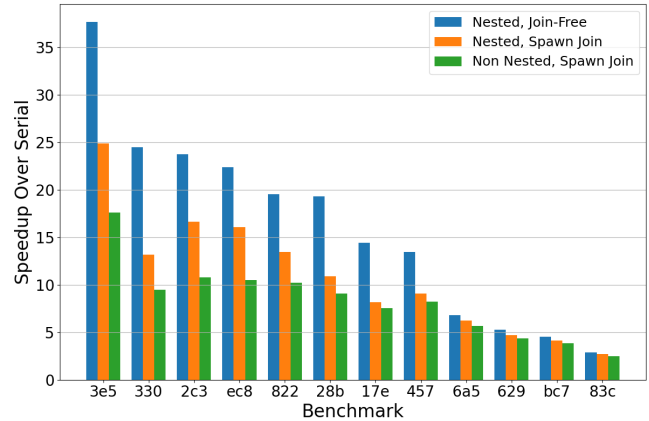
Once again, to gain some further insights on the sublinear scaling, we measured the data cache miss ratios obtained for memory accesses issued by the TRSs when varying the number of TRSs. Figure 9 shows these results. We can see that the data cache miss ratios increase steadily as we increase the number of TRSs. Our analysis shows that this increase is due to two reasons: (1) as more

TRSs try to parallelly access different data items (clauses), more conflicts arise for cache space, (2) as more TRSs try to parallelly access an item (clause) not present in cache memory, all of them can end up being cache misses.

4.4 Comparison of Different Parallelization Models



(a) Speedup obtained over the Serial Model for Different Parallelization Models, with 10 TEUs and 50 TRSs



(b) Speedup obtained over the Serial Model for Different Parallelization Models, with 50 TEUs and 250 TRSs

Figure 10: Comparison of speedup obtained over the Serial Model for different parallelization models with varying TEUs and TRSs.

The experimental results presented so far show significant speedup potential for our Join-Free Nested Spawn parallelization model. Lastly, we investigate how our model compares with the other two parallelization models that we discussed in this paper: traditional Spawn-Join model (Algorithm 3) and Nested Spawn-Join model (Algorithm 4). Figure 10 presents the speedup numbers we obtained for all three parallelization models over the serial algorithm; Figure 10 (a) shows the speedups obtained with 10 TEUs and figure10 (b) shows the speedups obtained with 50 TEUs. The X-axis plots the

benchmarks and the Y-axis plots the speedup numbers. Each benchmark has 3 bar charts, corresponding to the speedup obtained by (i) our Join-Free Nested Spawn model, (ii) the Nested Spawn-Join model, and (iii) the Non-Nested Spawn-Join model. These results show that our Join-Free Nested Spawn model consistently outperforms the other two parallelization models. *We view the speedup advantage of the Join-Free Nested Spawn over the Non-Nested one as the main quantitative evidence supporting the ideas in the current paper*, validating our effort to impose the least restrictions on concurrency towards exploiting whatever limited parallelism is present in the applications. Comparing Figure 10 (a) and (b), we can also see that the performance advantage of the Join-Free nested spawn model improves as more parallel hardware is made available. However, due to its diminishing scaling, as can be seen in Figure 6, we believe that our NPA is best used in conjunction with other parallelization techniques if the workload has different forms of parallelism than those considered in the current paper, such as higher amounts of parallelism, regular parallelism, and coarse-grained parallelism.

5 Conclusion

We believe that the job of a general-purpose parallel computer architecture is to provide the best performance for whatever forms of parallelism are available in an application. The key contribution of the current paper is in providing new architecture ideas for forms of parallelism that have not been sufficiently addressed in past work.

The parallelism in many important applications, such as SAT solvers, is mainly or exclusively irregular, which makes it difficult to exploit using traditional multi-cores or traditional accelerators such as GPUs. Our proposed Join-Free Nested Threading model and lightweight integrated NPA are engineered to exploit the irregular parallelism in such applications. Our experimental studies with a detailed microarchitectural simulator show high speedups and scalability with this software-hardware parallelization approach.

Owing to decades of work by many researchers the impact potential for scalable parallelism is enormous. Going forward, a "pull" from applications is key for realizing this potential: many-core and other scalable computing platforms need to find their major application "calling". This paper focuses on SAT solving, a potential killer app. Industry adoption of our architecture ideas can lead to new parallelization approaches for other apps, much in the same way that applications have always been the prime mover in advancing CPUs and HPC.

Acknowledgments

This work is partially supported by NSF grant 2427318.

References

- [1] Gilles Audemard and Laurent Simon. 2014. Lazy Clause Exchange Policy for Parallel SAT Solvers. In *Theory and Applications of Satisfiability Testing – SAT 2014*, Carsten Sinz and Uwe Egly (Eds.). Springer International Publishing, Cham, 197–205.
- [2] Gilles Audemard and Laurent Simon. 2018. On the Glucose SAT Solver. *International Journal on Artificial Intelligence Tools* 27, 01 (2018), 1840001. arXiv:https://doi.org/10.1142/S0218213018400018 doi:10.1142/S0218213018400018
- [3] Aydin Balkan, Gang Qu, and Uzi Vishkin. 2006. Mesh-of-Trees and Alternative Interconnection Networks for Single Chip Parallel Processing (Extended Abstract). (06 2006).
- [4] Aydin O. Balkan, Gang Qu, and Uzi Vishkin. 2009. Mesh-of-Trees and Alternative Interconnection Networks for Single-Chip Parallelism. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17, 10 (2009), 1419–1432. doi:10.1109/TVLSI.2008.2003999
- [5] Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda (Eds.). 2023. *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*. Department of Computer Science, University of Helsinki, Finland.
- [6] Armin Biere. 2010. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. <https://api.semanticscholar.org/CorpusID:1087401>
- [7] Armin Biere. 2013. Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013. <https://api.semanticscholar.org/CorpusID:972178>
- [8] Armin Biere. 2018. P LINGELING , T REENGELING and Y AL SAT Entering the SAT Competition 2018. <https://api.semanticscholar.org/CorpusID:59470158>
- [9] Guy Blelloch. 2011. *NESL*. Springer US, Boston, MA, 1278–1283. doi:10.1007/978-0-387-09766-4_225
- [10] G. E. Blelloch. 1989. Scans as Primitive Parallel Operations. *IEEE Trans. Comput.* 38, 11 (Nov. 1989), 1526–1538. doi:10.1109/12.42122
- [11] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: an efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Santa Barbara, California, USA) (PPOPP '95)*. Association for Computing Machinery, New York, NY, USA, 207–216. doi:10.1145/209936.209958
- [12] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2008. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.* 12, 2, Article 10 (dec 2008), 38 pages. doi:10.1145/1455518.1455522
- [13] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138. doi:10.1109/JSSC.2016.2616357
- [14] Stephen A. Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing (Shaker Heights, Ohio, USA) (STOC '71)*. Association for Computing Machinery, New York, NY, USA, 151–158. doi:10.1145/800157.805047
- [15] Joseph W. Cutler, Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Keshu Hietala, Eleftherios Ioannidis, John Kastner, Anwar Mamat, Darin McAdams, Matt McCutchen, Neha Rungta, Emina Torlak, and Andrew M. Wells. 2024. Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 118 (April 2024), 28 pages. doi:10.1145/3649835
- [16] Alessandro Dal Palù, Agostino Dovier, Andrea Formisano, and Enrico Pontelli. 2014. CUDA@SAT: SAT solving on GPUs. *Journal of Experimental Theoretical Artificial Intelligence* 27 (12 2014), 1–24. doi:10.1080/0952813X.2014.954274
- [17] John D. Davis, Zhangxi Tan, Fang Yu, and Lintao Zhang. 2008. A practical reconfigurable hardware accelerator for boolean satisfiability solvers. In *2008 45th ACM/IEEE Design Automation Conference*. 780–785. doi:10.1145/1391469.1391669
- [18] Martin Davis, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7 (jul 1962), 394–397. doi:10.1145/368273.368557
- [19] Rina Dechter. 2003. *Constraint Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- [20] James Edwards and Uzi Vishkin. 2021. Study of Fine-grained Nested Parallelism in CDCL SAT Solvers. *ACM Trans. Parallel Comput.* 8, 3, Article 17 (sep 2021), 18 pages. doi:10.1145/3470639
- [21] Niklas Eén and Niklas Sörensson. 2006. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation* 2, 1-4 (2006), 1–26. doi:10.3233/SAT190014
- [22] Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. 2017. PainleSS: A Framework for Parallel SAT Solving. In *International Conference on Theory and Applications of Satisfiability Testing*. <https://api.semanticscholar.org/CorpusID:32448463>
- [23] Zhaohui Fu and Sharad Malik. 2006. On solving the partial MAX-SAT problem. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (Seattle, WA) (SAT'06)*. Springer-Verlag, Berlin, Heidelberg, 252–265. doi:10.1007/11814948_25
- [24] Hironori Fujii and Noriyuki Fujimoto. 2012. GPU Acceleration of BCP Procedure for SAT Algorithms. <https://api.semanticscholar.org/CorpusID:60348757>
- [25] Weiwei Gong and Xu Zhou. 2017. A survey of SAT solver. *AIP Conference Proceedings* 1836, 1 (06 2017), 020059. arXiv:https://pubs.aip.org/aip/acp/article-pdf/doi/10.1063/1.4981999/13742154/020059_1_online.pdf doi:10.1063/1.4981999
- [26] Hariprasath Govindasamy, Babak Esfandiari, and Paulo Garcia. 2023. FPGAs (Can Get Some) SATisfaction. arXiv:2312.11279 [cs.AR] <https://arxiv.org/abs/2312.11279>
- [27] Kanupriya Gulati, Mandar Waghmode, Sunil P. Khatri, and Weiping Shi. 2008. Efficient, scalable hardware engine for Boolean satisfiability and unsatisfiable core extraction. *IET Comput. Digit. Tech.* 2 (2008), 214–229. <https://api.semanticscholar.org/CorpusID:2432795>

- [28] Aarti Gupta, Malay K. Ganai, and Chao Wang. 2006. SAT-Based verification methods and applications in hardware verification. In *Proceedings of the 6th International Conference on Formal Methods for the Design of Computer, Communication, and Software Systems (Bertinoro, Italy) (FDM'06)*. Springer-Verlag, Berlin, Heidelberg, 108–143. doi:10.1007/11757283_5
- [29] Youssef Hamadi and Christoph M. Wintersteiger. 2013. Seven Challenges in Parallel SAT Solving. *AI Mag.* 34, 2 (June 2013), 99–106. doi:10.1609/aimag.v34i2.2450
- [30] John L. Hennessy and David A. Patterson. 2017. *Computer architecture: a quantitative approach* (6th ed.). Morgan Kaufmann Publishers.
- [31] Troels Henriksen, Niels G. W. Serup, Martin Elsmann, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. *SIGPLAN Not.* 52, 6 (jun 2017), 556–571. doi:10.1145/3140587.3062354
- [32] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. 2011. Cube and conquer: guiding CDCL SAT solvers by lookaheads. In *Proceedings of the 7th International Haifa Verification Conference on Hardware and Software: Verification and Testing (Haifa, Israel) (HVC'11)*. Springer-Verlag, Berlin, Heidelberg, 50–65. doi:10.1007/978-3-642-34188-5_8
- [33] Daniel Kroening and Ofer Strichman. 2018. *Decision Procedures: An Algorithmic Point of View* (2nd ed.). Springer. <https://www.decision-procedures.org/>
- [34] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. 2016. Parallel graph analytics. *Commun. ACM* 59, 5 (apr 2016), 78–87. doi:10.1145/2901919
- [35] Mark H. Liffiton and Karem A. Sakallah. 2008. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *Journal of Automated Reasoning* 40, 1 (2008), 1–33. doi:10.1007/s10817-007-9084-z
- [36] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adria Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jayapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikolieris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglino, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. arXiv:2007.03152 [cs.AR] <https://arxiv.org/abs/2007.03152>
- [37] Salem Malikić, Simone Ciccolella, Farid Rashidi Mehrabadi, Camir Ricketts, Khaleddur Rahman, Ehsan Haghshenas, Daniel N. Seidman, Faraz Hach, Iman Hajira-souliha, and S. Cenk Sahinalp. 2018. PhISCS - A Combinatorial Approach for Sub-perfect Tumor Phylogeny Reconstruction via Integrative use of Single Cell and Bulk Sequencing Data. *bioRxiv* (2018). <https://api.semanticscholar.org/CorpusID:53550998>
- [38] Norbert Manthey. 2011. *A More Efficient Parallel Unit Propagation*. Technical Report. TU Dresden, Knowledge Representation and Reasoning Group.
- [39] Joao Marques-Silva. 2008. Practical applications of Boolean Satisfiability. In *2008 9th International Workshop on Discrete Event Systems*. 74–80. doi:10.1109/WODES.2008.4605925
- [40] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2015. High-Performance and Scalable GPU Graph Traversal. *ACM Trans. Parallel Comput.* 1, 2, Article 14 (feb 2015), 30 pages. doi:10.1145/2717511
- [41] Quirin Meyer, Fabian Schönfeld, Marc Stamminger, and Rolf Wanka. 2010. 3-SAT on CUDA: Towards a massively parallel SAT solver. In *2010 International Conference on High Performance Computing Simulation*. 306–313. doi:10.1109/HPCS.2010.5547116
- [42] Ilya Mironov and Lintao Zhang. 2006. Applications of SAT solvers to cryptanalysis of hash functions. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (Seattle, WA) (SAT'06)*. Springer-Verlag, Berlin, Heidelberg, 102–115. doi:10.1007/11814948_13
- [43] E. Mohr, D.A. Kranz, and R.H. Halstead. 1991. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems* 2, 3 (1991), 264–280. doi:10.1109/71.86103
- [44] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference (Las Vegas, Nevada, USA) (DAC '01)*. Association for Computing Machinery, New York, NY, USA, 530–535. doi:10.1145/378239.379017
- [45] Alexander Nadel. 2018. Solving MaxSAT with Bit-Vector Optimization. In *Theory and Applications of Satisfiability Testing – SAT 2018: 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9–12, 2018, Proceedings* (Oxford, United Kingdom). Springer-Verlag, Berlin, Heidelberg, 54–72. doi:10.1007/978-3-319-94144-8_4
- [46] Alexander Nadel and Vadim Ryzhkin. 2016. Bit-Vector Optimization. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636*. Springer-Verlag, Berlin, Heidelberg, 851–867. doi:10.1007/978-3-662-49674-9_53
- [47] oneTBB development Team. 2025. *oneAPI Threading Building Blocks (oneTBB)*. Retrieved February 25, 2025 from <https://www.threadingbuildingblocks.org/>
- [48] Muhammad Osama and Anton Wijs. 2019. Parallel SAT Simplification on GPU Architectures. In *Tools and Algorithms for the Construction and Analysis of Systems, Tomáš Vojnar and Lijun Zhang (Eds.)*. Springer International Publishing, Cham, 21–40.
- [49] Soowang Park, Jae-Won Nam, and Sandeep K. Gupta. 2021. HW-BCP: A Custom Hardware Accelerator for SAT Suitable for Single Chip Implementation for Large Benchmarks. In *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 29–34.
- [50] Roberto Sebastiani and Silvia Tomasi. 2012. Optimization in SMT with $\mathcal{L}\mathcal{A}(\mathbb{Q})$ Cost Functions. In *Automated Reasoning*, Bernhard Gramlich, Dale Miller, and Uli Sattler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 484–498.
- [51] Yian Su, Mike Rainey, Nick Wanninger, Nadham Dhiantravan, Jasper Liang, Umot A. Acar, Peter Dinda, and Simone Campanoni. 2024. Compiling Loop-Based Nested Parallelism for Irregular Workloads. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (La Jolla, CA, USA) (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 232–250. doi:10.1145/3620665.3640405
- [52] Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. 2014. Lazy Scheduling: A Runtime Adaptive Scheduler for Declarative Parallelism. *ACM Trans. Program. Lang. Syst.* 36, 3, Article 10 (sep 2014), 51 pages. doi:10.1145/2629643
- [53] Uzi Vishkin. 2003. Prefix sums and an application thereof. Filed November 1996.
- [54] Uzi Vishkin, Shlomit Dascal, Efraim Berkovich, and Joseph Nuzman. 1998. Explicit multi-threading (XMT) bridging models for instruction parallelism (extended abstract). In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (Puerto Vallarta, Mexico) (SPAA '98)*. Association for Computing Machinery, New York, NY, USA, 140–151. doi:10.1145/277651.277680
- [55] Tjark Weber. 2006. Integrating a SAT Solver with an LCF-style Theorem Prover. *Electronic Notes in Theoretical Computer Science* 144, 2 (2006), 67–78. doi:10.1016/j.entcs.2005.12.007 Proceedings of the Third Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR 2005).
- [56] Xingzhi Wen and Uzi Vishkin. 2008. Fpga-based prototype of a front-on-chip processor. In *Proceedings of the 5th Conference on Computing Frontiers (Ischia, Italy) (CF '08)*. Association for Computing Machinery, New York, NY, USA, 55–66. doi:10.1145/1366230.1366240
- [57] Chenzhuo Zhu, Alexander Rucker, Yawen Wang, and William J. Dally. 2023. SatIn: Hardware for Boolean Satisfiability Inference. *ArXiv abs/2303.02588* (2023). <https://api.semanticscholar.org/CorpusID:257365741>