Don't Repeat Yourself! Coarse-Grained Circuit Deduplication to Accelerate RTL Simulation

Haoyuan Wang
University of California, Santa Cruz
Santa Cruz, CA 95064, USA
hwang208@ucsc.edu

Thomas Nijssen
University of California, Santa Cruz
Santa Cruz, CA 95064, USA
tnijssen@ucsc.edu

Scott Beamer
University of California, Santa Cruz
Santa Cruz, CA 95064, USA
sbeamer@ucsc.edu

Abstract

Designing a digital integrated circuit requires many register transfer level (RTL) simulations for design, debugging, and especially verification. To cope with the slow speed of RTL simulation, industry frequently uses private server farms to run many simulations in parallel. Surprisingly, the implications of parallel runs of different RTL simulations have not been extensively explored. Moreover, in modern digital hardware, there is a growing trend to replicate components to scale out. However, the potential for circuit deduplication has been mostly overlooked.

In this work, we pinpoint the shared last-level cache as the primary bottleneck impacting the throughput of RTL simulation. To address this issue, we propose a coarse-grained circuit deduplication strategy integrated into an RTL simulator. Our method involves identifying multiple instances of a single module within a digital circuit and creating shared code that can be applied to all of these instances. Our approach reduces the cache footprint by increasing code reuse, which consequently benefits processor components such as caches and branch predictors. Our experiments demonstrate that deduplication can bring up to 1.95× speedup in a single simulation, and achieve up to 2.09× overall RTL simulation throughput.

CCS Concepts: • **Hardware → Hardware description languages and compilation**.

Keywords: RTL simulation, simulation throughput, deduplication

ACM Reference Format:
Haoyuan Wang, Thomas Nijssen, and Scott Beamer. 2024. Don't Repeat Yourself! Coarse-Grained Circuit Deduplication to Accelerate RTL Simulation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3622781.3674184

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0391-1/24/04
https://doi.org/10.1145/3622781.3674184

1 Introduction

The rapidly growing size and complexity of ASIC designs is causing the verification effort to grow at an exponential pace [18]. To ensure quality and timely delivery, chip companies generally maintain private compute infrastructure [14, 15] but naturally cannot scale this out as quickly as today's verification tasks require. Despite the growing need for high throughput RTL simulation, the implication of batch execution of different RTL simulators is underexplored.

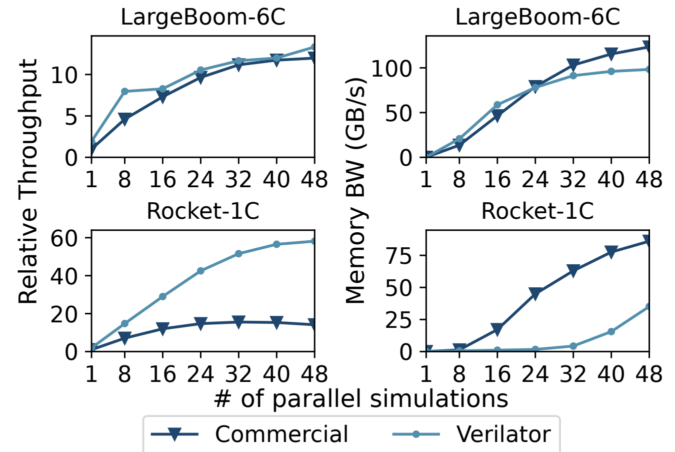


Figure 1. Throughput scaling limits for independent RTL simulations running in parallel on a large (LargeBoom-6C) and a small (Rocket-1C) design. Throughput normalized to a single commercial simulator. Evaluation details in Section 6.

Despite their independence, in practice, a batch of simulations does not often demonstrate linear parallel scaling (Figure 1). Our motivating experiment considers leading RTL simulators, both commercial and open-source (details in Section 6). For a large design (LargeBoom-6C), 48 physical cores executing 48 independent simulations, both simulators achieve around 12× throughput increase. When more simulations execute in parallel but the overall throughput plateaus, that implies each simulation is taking longer. The simulation slowdown is due to contention for shared resources (last-level cache and off-chip memory bandwidth).

When considering a smaller design under the same conditions (Rocket-1C), we observe Verilator scales much better than the commercial simulator. With the smaller design, Verilator is able to keep much of it in cache, and thus it only needs a small fraction of the system's memory bandwidth. This motivating experiment demonstrates the potential of shrinking the cache footprint of simulation to greatly increase scalability.

The replication in modern hardware designs provides a great opportunity to reduce the cache footprint of simulation. Modern designs are rarely composed of unique singleton modules. Parallelism is frequently exploited, whether it be for multiple tiles, cores, vector lanes, etc. Contemporary ASIC projects extensively leverage existing building blocks verified by third-party companies [10], which can also result in replication. This practice not only alleviates the verification burden and expedites product development, but it also opens the door to strategic optimization.

These replicated components can be *deduplicated* such that the simulator reuses infrastructure for all repeated instances. Each instance will have its own data, but they can share the code that represents the instances, which is typically much more problematic than the data. Successful deduplication relieves the high pressure placed on the last-level cache and in turn enhances overall simulation throughput. Increased throughput can help improve verification coverage, reduce time to market, and even reduce design costs.

While modern hardware design extensively leverages building block reuse, applying deduplication in RTL simulation presents unique challenges, distinct from those in software development. There are usually dependencies between a duplicated component and its surrounding context such that simply creating a single shared function for the duplicated module will almost certainly create a scheduling deadlock. Overcoming this deadlock by repeatedly evaluating components introduces extra computation. Thus, the primary challenge is breaking up a deduplicated component into portions that can be reused while being efficiently scheduled such that each partition is evaluated at most once.

In this work, we explore the challenges for deduplication, our solution to these challenges, and analyze when and why it is beneficial to deduplicate for simulation. We introduce a method to generate code for a single instance that can be shared for multiple duplicated instances in the circuit design. By carefully scheduling the simulator's execution, we create frequently re-executed code regions, forming beneficial hotspots. Our techniques reduce the effective cache footprint of RTL simulation, which greatly increases the scalability of batch simulation throughput. We contribute:

- We identify the shared last-level cache as the primary bottleneck for the throughput of multiple simultaneous RTL simulations, for both commercial and open source simulators.
- We enable deduplication in RTL simulation by eliminating cycles in the circuit's partitioned graph.
- We propose a method to co-schedule deduplicated components to increase code locality which we demonstrate improves branch prediction effectiveness.
- Deduplicated simulation can lead up to a 1.95× speedup in a single simulation, and achieve 2.09× overall simulation throughput compared to our baseline which does not perform deduplication. Depending on the design, deduplication reduces the graph partitioning time by up to 5.68× compared to our baseline.

2 Background

2.1 RTL Simulation Overview

Register transfer level (RTL) simulation serves as a foundational step in digital hardware design, allowing engineers to validate the functionality and correctness of their designs before they are translated into actual hardware. At this level, the behavior of the digital system is described in terms of data movement between registers and logic operations with clock cycle-level fidelity.

There are typically two ways to simulate a design: *event-driven*, in which every time a signal in the design changes, it emits an event to everything connected to it (which, in turn, will cause another signal to change, emitting more events, and so on); and *full-cycle* which unconditionally evaluates the new state of the entire design without considering whether or not a signal has changed. Event-driven simulation matches the specification semantics of hardware description languages such as Verilog, but it comes with a high scheduling overhead determining which event to evaluate next. Full-cycle simulation effectively inlines the entire design to create a custom simulator to remove that scheduling overhead. To first order, one can think of event-driven simulation as an interpreter (hardware design as data), and full-cycle simulation as ahead-of-time compilation (hardware design as code). Both simulation approaches are commonly used, however, full-cycle simulators tend to be the fastest [3, 5, 20].

Verilator is a high-performance Verilog simulator [20] that uses a full-cycle approach. It is commonly used in industry and academia because of its simulation speed and that it is free and open source. We compare against Verilator in our evaluation due to its availability and widespread use.

ESSENT is an open-source RTL simulator whose research contribution is exploiting low-activity factors in the design to avoid unnecessary computation [5]. Although its optimizations make it a hybrid, it is structurally full-cycle, as it compiles the hardware design into code ahead of time. We extend ESSENT to prototype our approach due to its small modular codebase that eases our development. Our deduplication work also leverages ESSENT's acyclic partitioner.

2.2 Importance of Cache for RTL Simulation

As a workload, RTL simulation is largely instruction bound. Event-driven simulators execute many instructions to determine the next portions of the hardware to evaluate. Full-cycle simulators (such as Verilator and ESSENT) have simulator binaries that grow proportionally with the design size. Although the execution is largely straight line, these large programs still overwhelm the host processor's frontend (caches and branch predictors). This frontend bottleneck currently restricts simulators to modest instruction throughput rates.

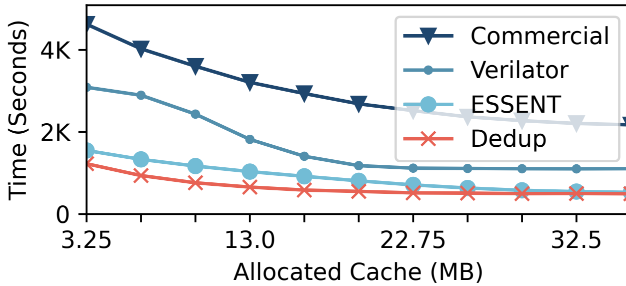


Figure 2. All of the evaluated simulators slow down on a large design (LargeBoom-6C) when the last-level cache (LLC) capacity is constrained using Intel's RDT on our server (Table 1). For a single simulation, there is diminishing returns when allocating more LLC capacity to the simulator.

To appreciate the cache's importance, we limit the last-level cache (LLC) available to each simulator and observe the great increase in simulation time when there is insufficient LLC capacity (Figure 2). Our observation that RTL simulation is particularly cache-hungry is consistent with prior work [7, 22] and helps to explain the performance plateau from our motivating parallel scaling experiment (Figure 1). When multiple RTL simulators run on a single server, they consume more cache than is available on the host system, creating substantial memory bandwidth demand. As a result, with insufficient cache capacity, cores are left idle as they wait for memory to be retrieved. Executing even more simulations in parallel not only does not solve this problem, it can even make it worse.

2.3 RTL Simulation in Industry

Due to substantial computational demands for electronic design automation (EDA), chip companies traditionally have their own private computing infrastructure [12, 14, 15]. The potential throughput penalty when running multiple RTL simulators in parallel on a single server has become a point of concern in industry [21], however the issue remains underinvestigated. Some experiments demonstrate near-linear scaling of RTL simulation throughput [15], while other studies conclude to the contrary that most cores should be idle to preserve overall throughput [21].

CPU vendors have long appreciated that providing a large LLC benefits many applications, including EDA software. For example, AMD processors with hybrid-bonded cache (3D V-Cache) [25] greatly increase the LLC capacity per core and bring a 66% simulation speedup in some cases [17]. However, our evaluation demonstrates that RTL simulation is extremely cache-hungry, to the extent that the 3D V-Cache only partially alleviates the cache stress (Figure 10).

2.4 Existing Deduplication in RTL Simulators

Current RTL simulators offer limited deduplication support. Recent academic simulators [5, 22, 27] lack deduplication capabilities. In contrast, Verilator [20], a widely-used open-source simulator, implements deduplication for small SystemVerilog statements. However, our evaluation reveals that the full potential of deduplication remains significantly untapped (Section 6). We cannot be sure about how the commercial simulator is exactly implemented, but our analysis suggests either deduplication is not performed or at least is ineffective.

2.5 Graph Representation & Partitioning

A digital circuit can be represented as a directed graph, with gates and registers being the nodes, and wires as the edges. Most simulators leverage this graph representation as a useful abstraction for developing infrastructure such as an internal intermediate representation (IR). A nice attribute of hardware is that the dataflow is completely known statically, which is significantly simpler than the case of arbitrary software.

Cycle-accurate RTL simulators naturally simulate cycles in order, one-at-a-time. Thus, much of the engineering effort focuses on speeding up the evaluation of a single cycle. In that cycle, new values emerge from registers, memories, and external inputs, flow through logic elements, and are stored into registers, memories, and external outputs for the next cycle. To perform this evaluation efficiently, simulators aspire to evaluate each element at most once per simulated cycle. Evaluating elements in a breadth-first manner (levelized) solves this problem [23, 24], but it eliminates potentially beneficial producer-consumer temporal locality across levels. Full-cycle simulators can generate an efficient schedule statically with an algorithm such as topological sort, but that requires the design graph to be *acyclic.* Fortunately, most hardware designs are acyclic or nearly acyclic. A nearly acyclic design can be made acyclic by grouping strongly-connected components into supernodes.

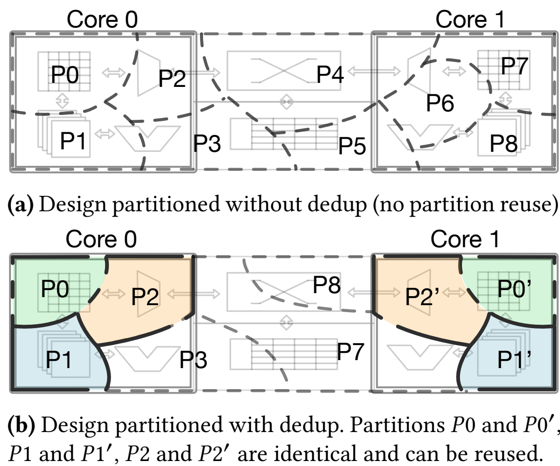
RTL simulators, particularly full-cycle simulators, frequently partition the design. Breaking up the circuit into smaller subgraphs allows for modest sized partitions which can benefit much of the tool flow. For example, when translating the design into code to be compiled, reducing the size of functions can greatly speed up compilation [20]. Alternatively, well-chosen partitions can ease memoization optimizations

to exploit low-activity factors [5]. Partitions can even be used to statically allocate work to threads for multicore parallelization [22]. Partitioning a hardware design graph can often be complicated by acyclic constraints. An acyclic partitioning allows for efficient simulation in which each partition is evaluated at most once per simulated cycle.

3 Motivation & Challenges

3.1 Motivation: Deduplicate for Code Reuse

In the realm of modern ASIC development, the efficiency of RTL simulation throughput is profoundly influenced by the available Last-Level Cache (LLC) size. Naturally, reducing the code footprint stands out as a pivotal strategy to enhance overall simulation throughput. By generating reusable code once and executing it multiple times, a remarkable reduction in the simulator's code footprint can be achieved. This reduction, in turn, leads to a substantial enhancement in throughput (Figure 3).



(a) Design partitioned without dedup (no partition reuse)

(b) Design partitioned with dedup. Partitions $P0$ and $P0'$, $P1$ and $P1'$, $P2$ and $P2'$ are identical and can be reused.

Figure 3. Partitioning an example SoC with two identical instances (core 0 and core 1, in doubled line). Without deduplication (Figure 3a), arbitrary partitions can cross instance boundaries and none of the code is reused . With our deduplication method (Figure 3b), partitions within replicated modules are identical so the code can be reused (same color partitions).

To capitalize on performance and throughput gains, it is essential to focus on deduplicating a significant portion of the input circuit. To fulfill this objective, we adopt a coarse-grained approach to deduplication. This method involves identifying multiple instances of the same module and maximizing the reuse of partitions wherever feasible. By concentrating on this coarse-grained level of deduplication, we aim to optimize performance and enhance overall simulation throughput effectively.

To successfully implement deduplication in existing RTL simulators and achieve favorable outcomes, we have identified the following key challenges:

3.2 Challenge 1: Avoiding Potential Cycles

In order to enjoy the code reuse benefits of deduplication, that piece of generated code must be reusable in multiple instances. Although a replicated module and its repeated instances can be readily identified from the module hierarchy, the module boundaries are typically ill-suited for code reuse. Modules typically have bidirectional connections, so modules directly translated into partitions will introduce cycles.

Additionally, different instances of the same module may have different connectivities in their surrounding contexts. A partitioning of the module may be acyclic in one context, but not another. Figure 4 provides an example, while a legal, acyclic partitioning can be easily applied to one instance (Figure 4a), that partitioning applied to a different instance introduces a cycle due to an additional external connection (Figure 4b).

Cycles in the partitioning graph pose a significant challenge in RTL simulation, as they can lead to scheduling deadlocks. A scheduling deadlock can be broken by repeatedly evaluating components until a steady state is achieved. However, this method to break a deadlock leads to substantial inefficiency due to executing extra instructions from evaluating some components multiple times. For this reason, a novel duplication-aware partitioning strategy is needed. This approach must proactively prevent the formation of cycles, or promptly resolve them if they emerge.

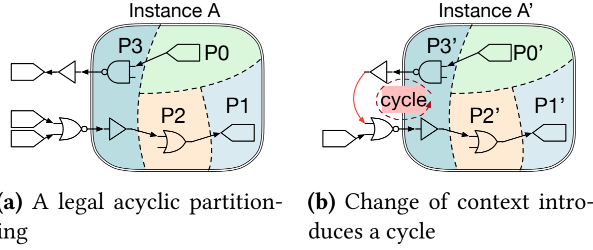
3.3 Challenge 2: Performance Degradation Due to Indirect Memory Accesses

Deduplication, if successful, will increase instruction throughput through code reuse. However, refactoring the generated code to support reuse will inevitably increase the number of instructions executed. This is especially true for signals at the boundary of the reused code.

In RTL simulators without deduplication, the locations of signals read and written can be "hardcoded" per partition, a technique adopted in prior research [5, 20]. Providing this information statically during compilation enables extensive optimizations by the compiler, including even obviating the need to use memory for some signals.

Regrettably, deduplication disrupts this optimization opportunity. It necessitates memory indirection for every signal into or out of the deduplicated partition. Consequently, the memory addresses are no longer readily available for the compiler, preventing certain optimizations which sacrifice some performance.

Indirect memory accesses can impose additional burdens. While modern processors are superscalar, they must still process more micro-ops to handle indirect accesses than they would have to for direct accesses. Additionally, the compiler



(a) A legal acyclic partitioning **(b)** Change of context introduces a cycle

Figure 4. Reusing a partitioning in another instance may lead to cycles in the graph due to differing graph contexts. In this example, when the acyclic partitioning from instance A is directly applied to instance A', different paths within the partitioning are connected by an external edge. Although the circuit graph itself remains acyclic, this newly introduced external edge creates a cycle involving partition P3.

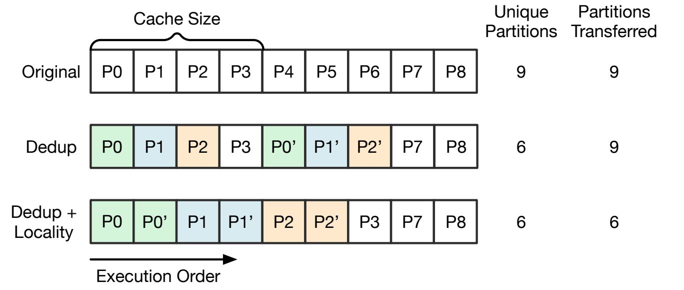


Figure 5. By emitting reusable code for identical partitions, deduplication substantially decreases the number of unique partitions. Additionally, locality-aware scheduling significantly reduces the number of partitions transferred through the memory hierarchy by enhancing temporal locality.

is forced to emit more instructions for a given block due to optimizations prevented by the indirection. These additional instructions eat into scarce instruction cache capacity.

We attribute this degradation to the overhead of indirect accesses, and we refer to it as the "dedup tax". In summary, executing more instructions and larger code footprints can undermine the benefits deduplication can provide. Thus, a sufficient portion of the design must be able to be deduplicated to overcome this overhead.

3.4 Challenge 3: Data Layout

Deduplication mandates indirect accesses to instance-specific data structures, thereby imposing new constraints on the data layout. For example, reused code must be able to accept inputs from all of its instances as well as produce outputs that the surrounding context can use. Thus, the code surrounding the reused code must adjust its data layout to accommodate the input or output layout expected by the reused code.

3.5 Opportunity: Code Locality

In RTL simulation without deduplication support, code will be generated for every partition; identical hardware instances within the circuit design will result in redundant, mostly repetitive code for each one. This redundancy leads to poor resource utilization, as the system will be limited by the speed of fetching the code for each partition. However, with deduplication, the simulator can achieve mostly identical partitioning for these instances. This facilitates code reuse for identical partitions, significantly enhancing efficiency, as depicted in Figure 5.

Furthermore, this code reuse presents a novel optimization opportunity in partition scheduling. Scheduling duplicated partitions and their counterparts in other instances (as exemplified by partitions $P0$ and $P0'$, $P1$ and $P1'$, $P2$ and $P2'$ in Figure 3 and Figure 5) can significantly enhance temporal code locality to help compensate for the performance loss caused by indirect accesses (Challenge 2 in Section 3.3).

However, this promising scheduling strategy is not universally applicable for every deduplicated partition because there are inherent data dependencies that must be preserved. Consequently, the scheduler must minimize the scheduling distance between reused partitions while ensuring correct scheduling order to maximize temporal locality.

4 Partitioning for Deduplication

The core challenge to deduplication is partitioning the design such that the deduplicated code can be reused, while also keeping the graph of partitions acyclic (Challenge 1 in Section 3.2). We propose a pragmatic partitioning method robust to the potential cycles introduced by deduplication that is sufficiently fast to be practical. In summary, it sacrifices some of the periphery around the deduplicated module instances to ease maintaining the acyclic constraint.

First, the tool flow must first identify duplicate modules to deduplicate. Duplicate instances can be readily identified from the module hierarchy by examining modules that are instantiated multiple times. When considering which repeated module to deduplicate, there are several viable approaches (Figure 6). In this work, we focus on deduplicating a single module and its instances (Figure 6a), specifically choosing the module that offers the maximum benefit (calculated as the product of the number of module instances and the module's size). Extending our methodology to handle multiple sets or even nesting could further increase the benefit of deduplication, but our simplification allows us to focus on the key factors.

4.1 Our Adaptive Partitioning Approach

Our partitioning method opts for a stepwise strategy instead of attempting to create a universal acyclic partitioning applicable to all instances slated for deduplication right from

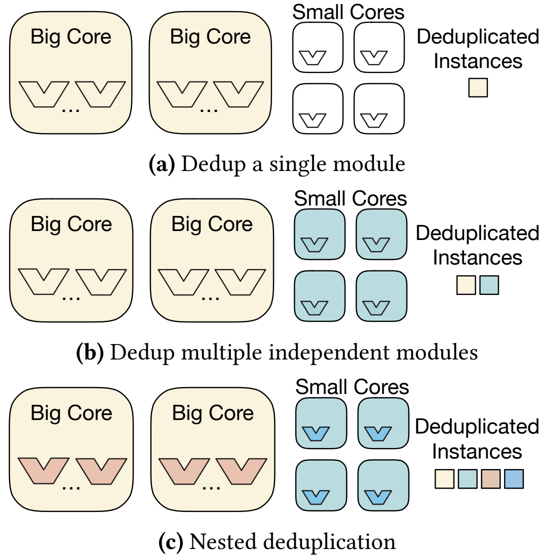


Figure 6. Potential approaches of circuit deduplication. The simple coarse-grained approaches are the easiest to implement, but the benefit of reducing the code size even further by deduplicating other modules as well must not be overlooked.

the start. We initially partition a single instance and then apply this partitioning to the remaining instances. However, a straightforward application of that partition to all instances is likely to introduce some cycles (Challenge 1 in Section 3.2), so we must be able to break cycles.

Unlike prior acyclic partitioning methods that address cycles by relocating vertices between partitions [11], we take a different route. We effectively eliminate these cycles by strategically *dissolving* partitions (breaking certain partitions back into vertices) and subsequently re-partitioning the graph remnants. As a result, the central partitions for the module are unmodified and can thus utilize reused code. The dissolved partitions on the periphery are modified, and thus unable to reuse code. Our selective approach allows us to adapt the acyclic partitioning to each instance's context.

4.2 Efficiently Selecting Partitions to Dissolve

Dissolving partitions that cause cycles, and subsequently re-partitioning, offers a robust solution to the cycle problem within partitioning for deduplication. In the worst-case, in which every partition in every deduplicated instance is dissolved, the problem reverts to the original partitioning challenge without deduplication. In such cases, subsequent acyclic partitioning naturally yields legal partitions. Such a situation might occur when no module has multiple instances, or when the module earmarked for deduplication is too small to exhibit a noticeable performance difference, making the deduplication effort unnecessary in the first place.

The computational cost of identifying cycles in the graph is acceptable, but we can significantly reduce it. Cycles that can potentially arise from the direct reuse of partitions across different instances stem from differing connectivities in their external contexts (e.g. $P3$ and $P3'$ in Figure 4). Thus, any cycles that arise will cross the instance boundary, as a cycle purely internal to the instance contradicts the instance being acyclically partitioned.

To leverage our insight, we simply dissolve all instances' partitions on the boundary. Dissolving these partitions has minimal impact on the deduplication rate. In practical circuit designs, large modules typically contain significantly more internal circuit than I/O signals. After dissolving all boundary partitions, it is still theoretically possible for cycles to persist, potentially encompassing internal partitions that are now on the fringe. Any such cycles could be resolved by iteratively dissolving partitions in the cycle. However, in our practical experimentation, dissolving only the boundary partitions is sufficient. Hardware designs are mostly acyclic, and by breaking a deduplicated instance into multiple partitions, there is already enough flexibility to avoid cycles. The potential for cycles may also be greatly reduced by the inherent characteristics of ESSENT's acyclic partitioner we use in this work which carefully coarsens partitions.

4.3 Deduplication Partitioning Method Summary

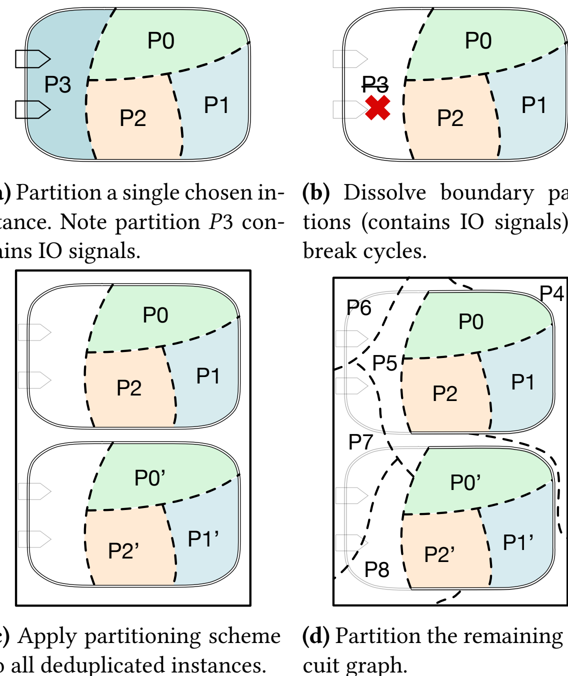


Figure 7. Adaptive partitioning approach for deduplication. Doubled line represents instance boundary, and dashed line indicates partition boundary.

Our partitioning algorithm takes the following steps:

1. **Partition a single instance (Figure 7a).** Once the module earmarked for deduplication is selected, acyclically partition a single instance of the designated module. This partitioning will serve as a template for all other instances.
2. **Repeatedly break partitions for every instance (Figure 7b).** For each instance, partitions causing loops are dissolved back into vertices. To maintain an acceptable partitioning speed, we adopt a cautious yet effective approach: initially break down all boundary partitions and verify no cycles exist. If any cycles are present, dissolve the partitions involved, and repeat. However, for the practical designs we study, we have not encountered any cycles after removing all boundary partitions.
3. **Apply partitioning scheme to all deduplicated instances (Figure 7c).** Once acyclic partitions have been determined for each instance and ensured that no loops exist, this partitioning scheme is uniformly applied to all instances in the original circuit graph.
4. **Partition the remaining circuit graph (Figure 7d).** Apply the acyclic partitioner to the remaining portion of the circuit's graph. This process yields an acyclic partitioning in which instances of the chosen module have mostly identical partitions that can reuse code.

4.4 Discussion of Our Method's Benefits

Our pragmatic method is able to produce acyclic partitions that are identical across instances (so code can be reused) while being practically efficient. We qualitatively discuss some key points:

- **Computationally efficient** - We partition one instance of a repeated module and are able to efficiently reuse a large fraction of that partitioning for the other instances. An alternative naïve approach might attempt to make a single partitioning that is safe to apply to all instances, but doing so requires traversing the large design graph (expensive) to create a unified constraint graph (to partition).
- **Partition size is only mildly important** - Full-cycle simulators do not strictly enforce partition size constraints since an imbalanced partitioning is acceptable. Therefore, the computational cost that a general acyclic partitioner expends to carefully relocate vertices to balance partitions is unnecessary.
- **Small reduction in deduplication coverage is tolerable** - Deduplicating the entirety of a module is not always essential, since each module can be partitioned into multiple smaller partitions. Opting to not to deduplicate a few of these partitions in exchange for faster compilation (including partitioning) and robustness to cycles is a worthwhile tradeoff. Our strategy prioritizes practicality and reduced compilation time while still enjoying most of the benefits of deduplication.

5 Data Layout & Scheduling

Beyond partitioning for deduplication, there are both data layout (Challenge 3 in Section 3.4) and scheduling considerations (Section 3.5) for applying deduplication, which can impact both correctness and performance.

5.1 Data Layout

To reuse the same code for different instances requires indirection for the data accesses. To ease that indirection, the data should have the same layout, so only the base pointer needs to change. For each deduplicated module, we employ a single `struct` per instance, which internally could support multiple deduplicated partitions. Those deduplicated partitions are designed for code reuse, and can leverage the addressing predictability of the struct. Partitions bordering with the deduplicated partition are modified to read or write this struct for signals going into or out of the deduplicated partitions. Thus, data accesses within a deduplicated partition or its boundary use a pointer, while all other accesses are direct. Our approach avoids the need for additional copy operations to support the transition from one-time-use code to reused code.

5.2 Temporal Locality-Aware Scheduling

A significant benefit of deduplication is the ability to activate a fraction of the code more than once per simulated cycle. This is particularly advantageous for the processor frontend and caches, especially when calls to the same code are closely spaced (Figure 5).

A correct execution order must respect the data dependencies in the partition graph. An acyclic partition graph may have multiple legal topological orders, but unfortunately, not all corresponding partitions can be scheduled together due to external dependencies between them. Thus, the main objective of our locality-aware scheduling is to efficiently find a legal topological order while scheduling reused code as close as possible.

Our locality-aware scheduling method:

Step 1: Consolidation: We consolidate corresponding partitions from multiple instances into a single super partition. We handle the challenge of avoiding cyclic merges by following the rule proposed by Herrmann et al. [11], further enhanced by Beamer and Donofrio [5]:

Theorem 5.1. *Partitions A and B can be safely merged $\iff$ there is no external path in either direction between them.*

Step 2: Topological Sorting: We apply topological sort on the consolidated graph to obtain a legal schedule that is a mixture of super partitions and regular partitions.

Step 3: Disassembly: Each super partition is disassembled, releasing its constituent partitions and allowing the individual partitions within it to be scheduled together.

6 Evaluation

6.1 Methodology

In this section, we present our evaluation of the deduplication and code generation methods implemented on top of ESSENT [5]. As with ESSENT, we accept FIRRTL [13] as the input language, and emit the simulator in C++. The evaluation encompasses the following simulators:

- **Commercial**: A widely used commercial simulator, anonymized due to licensing restrictions.
- **Verilator [20]**: An open-source RTL simulator embraced by both academia and industry [3]. Deduplication is enabled by default.
- **Verilator-NoDedup**: Verilator with dedup disabled.
- **ESSENT [5, 6]**: A high performance RTL simulator, serving as our baseline.
- **Dedup**: ESSENT, enhanced with deduplication partitioning and locality optimization techniques.

We additionally create two variants of our simulators to assess the effectiveness of specific optimizations:

- **PO** (**P**artitioning **O**nly): ESSENT using the new partitioning from our deduplication method without the corresponding code reuse (Section 4.3).
- **NL** (**N**o **L**ocality optimization): Dedup including code reuse but excludes the locality-aware scheduling optimization (Section 5.2).

We perform the evaluation on two platforms: Server and Desktop (Table 1). Server epitomizes a typical dual-socket server. In light of our emphasis on reducing the working set size, we include Desktop, which is equipped with an additional hybrid-bonded L3 cache [25], to scrutinize the performance impact on machines with ample cache capacity.

Table 1. Evaluation Platforms

Field	Server	Desktop
CPU	2× Intel Xeon Platinum 8260	1× AMD Ryzen 5800X3D
L1 Cache	48× 32 KB L1I, 32 KB L1D	8× 32 KB L1I, 32 KB L1D
L2 Cache	48× 1 MB L2	8× 512 KB L2
L3 Cache	2× shared 35.75 MB L3	1× 96 MB L3 (3D-VCache)
DRAM	2× 6 Ch. DDR4-2666 (250GB/s)	2 Ch. DDR4-3200 (50GB/s)
OS	Debian 12, Linux kernel 6.1.0	
Compiler	g++ 12.3.0, -O3	
Verilator	Verilator 5.016, -O2, -fno-dedup for Verilator-NoDedup	

We evaluate the simulators using 17 designs (Table 2) generated by the following design generators:

- **Rocket-*n*C**: *n* core Rocket Chip [1]. Rocket Chip is an open-source SoC generator featuring Rocket, an in-order RISC-V core.
- **Boom-*n*C**: *n* core Boom [26]. BOOM is a parameterizable open-source out-of-order RISC-V core written in the Chisel [2] hardware description language. For our evaluation, we utilize several common Boom configurations, including *SmallBoom* (1-wide with 32 ROB entries), *LargeBoom* (3-wide with 96 ROB entries), and *MegaBoom* (4-wide with 128 ROB entries).

We execute the vvadd (RISC-V vector-vector addition) microbenchmark on the simulated CPU designs.

Table 2 also offers insights into both the theoretical node reduction achievable if every node inside all duplicated instances could be successfully removed (deduplicated), and the actual node reduction accomplished by this work, which, due to the necessity to maintain acyclic structures, dissolves some partitions. In the context of single-core designs, our analysis reveals that only small modules can be deduplicated, resulting in only a minor reduction in nodes (up to 3.21%). Conversely, for multi-core designs, our approach effectively identifies and deduplicates the largest module, the multiple processor cores. With regards to the incurred cost of dissolving boundary partitions (Section 4.2), this is deemed acceptable, particularly when considering that components such as interconnect and peripherals are not subjected to deduplication.

Table 2. Evaluated Designs. The larger designs have more reuse and thus enable a greater node count reduction. Based on the size of the duplicated modules, we can calculate the reduction we could ideally get from deduplicating those. In reality, we cannot actually deduplicate all of those, since then the node graph would become cyclic.

Design	Nodes	Edges	Ideal Node Reduction	Real Node Reduction
Rocket-1C	61,974	119,538	1.04%	0.00%
Rocket-2C	90,071	175,386	29.06%	20.80%
Rocket-4C	147,181	289,422	53.34%	37.76%
Rocket-6C	207,034	407,933	63.20%	44.41%
Rocket-8C	263,035	519,681	69.65%	49.96%
SmallBoom-1C	105,441	236,919	1.01%	0.36%
SmallBoom-2C	176,949	410,098	42.06%	28.65%
SmallBoom-4C	323,881	763,727	68.94%	45.13%
SmallBoom-6C	466,501	1,109,782	79.77%	54.56%
SmallBoom-8C	615,504	1,466,036	84.64%	57.63%
LargeBoom-1C	211,087	520,192	6.44%	3.21%
LargeBoom-2C	389,204	977,664	47.33%	34.16%
LargeBoom-4C	744,807	1,892,150	74.19%	53.42%
LargeBoom-6C	1,092,878	2,793,537	84.27%	60.86%
LargeBoom-8C	1,453,992	3,716,309	88.68%	63.73%
MegaBoom-1C	312,149	789,130	4.35%	2.15%
MegaBoom-2C	590,314	1,513,832	48.29%	36.18%
MegaBoom-4C	1,146,400	2,962,724	74.60%	56.32%
MegaBoom-6C	1,688,307	4,387,733	84.43%	63.18%
MegaBoom-8C	2,256,909	5,856,867	88.42%	66.13%

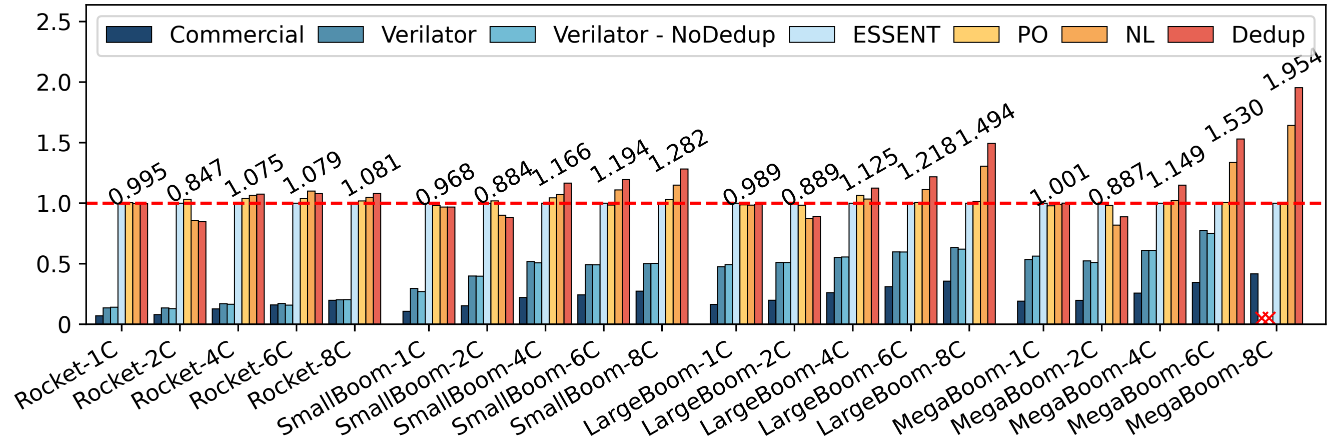


Figure 8. Relative simulation speed of a single simulation, normalized to ESSENT (red dashed line), on Server platform. Relative speedup of Dedup is labeled numerically. Large designs with a lot of reuse benefit most from the deduplication and locality enhancements; conversely, smaller designs or those without much reuse only see performance similar to the baseline. Verilator fails to compile `MegaBoom-8C`.

6.2 Benefits for Single RTL Simulation Performance

We first evaluate the performance of a single RTL simulation running in isolation, so each simulator is provided access to all of the L3 cache on a single socket of the server. The results, depicted in Figure 8, reveal that in the case of single-core designs which have limited deduplication potential (as outlined in Table 2)—the performance difference between the baseline (`ESSENT`) and `Dedup` is negligible.

However, as the simulated core count increases, simulation performance noticeably degrades with deduplication on all dual-core designs. The observed performance degradation is a predictable consequence of the dedup tax (Challenge 2, Section 3.3). Thus, deduplication is only warranted if there is sufficient replication in the design, to overcome the dedup tax.

As the simulated core count increases further, `Dedup` surpasses `ESSENT` in performance. This trend is expected, as increased duplication benefits `Dedup` by simultaneously reducing the simulation's working set size (thanks to fewer unique instructions in the binary) and amplifying the effectiveness of locality-aware scheduling. Notably, `Dedup` achieves its highest performance on the `MegaBoom-8C` configuration, outpacing the baseline by approximately 1.95×. `Dedup` yields even higher performance gains if the degree of replication increases (in this design, the number of processor cores instantiated).

Figure 8 further illustrates that across most tested designs, `PO` and `ESSENT` exhibit comparable simulation speeds. Although the partitionings produced by our new deduplication method differ from the original ESSENT partitionings, our experiments indicate the change in partitions is not the primary cause of `Dedup`'s performance advantage.

Similarly, the significance of locality-aware scheduling is underscored by the `NL` variant. In the absence of this optimization, the performance of the `NL` model exceeds that of the baseline but demonstrates a substantial deficiency when compared to the `Dedup`. It also performs worse in dual-core scenarios due to the dedup tax.

By default, Verilator enables deduplication, automatically processing certain SystemVerilog's *assign* statements and simple *always* statements, but will not attempt more complex statements. In contrast, our method is capable of deduplicating almost any kind of connectivity since `ESSENT` does not give special treatment to any particular kind of statements. As shown in Table 2, there can be a significant reduction in node count with deduplication in `Dedup`. This means that the usefulness of Verilator's deduplication is questionable, since it will only be possible in specific circumstances. Indeed, the performance discrepancy between `Verilator` and its variant without deduplication (`Verilator-NoDedup`) is negligible (Figure 8 & Figure 9).

6.3 Multiple RTL Simulation Throughput Boost

While the performance gain for an individual simulation is noteworthy, our primary contribution extends beyond the performance of a single simulation. In an industrial setting, the overall simulation throughput profoundly influences verification progress and comprehensive coverage. In this section, we assess the simulation throughput (in terms of simulated cycles per wall clock time) across the entire system.

To replicate a realistic environment closely resembling a real EDA datacenter, we concurrently run multiple simulations. In a production environment, individual simulations

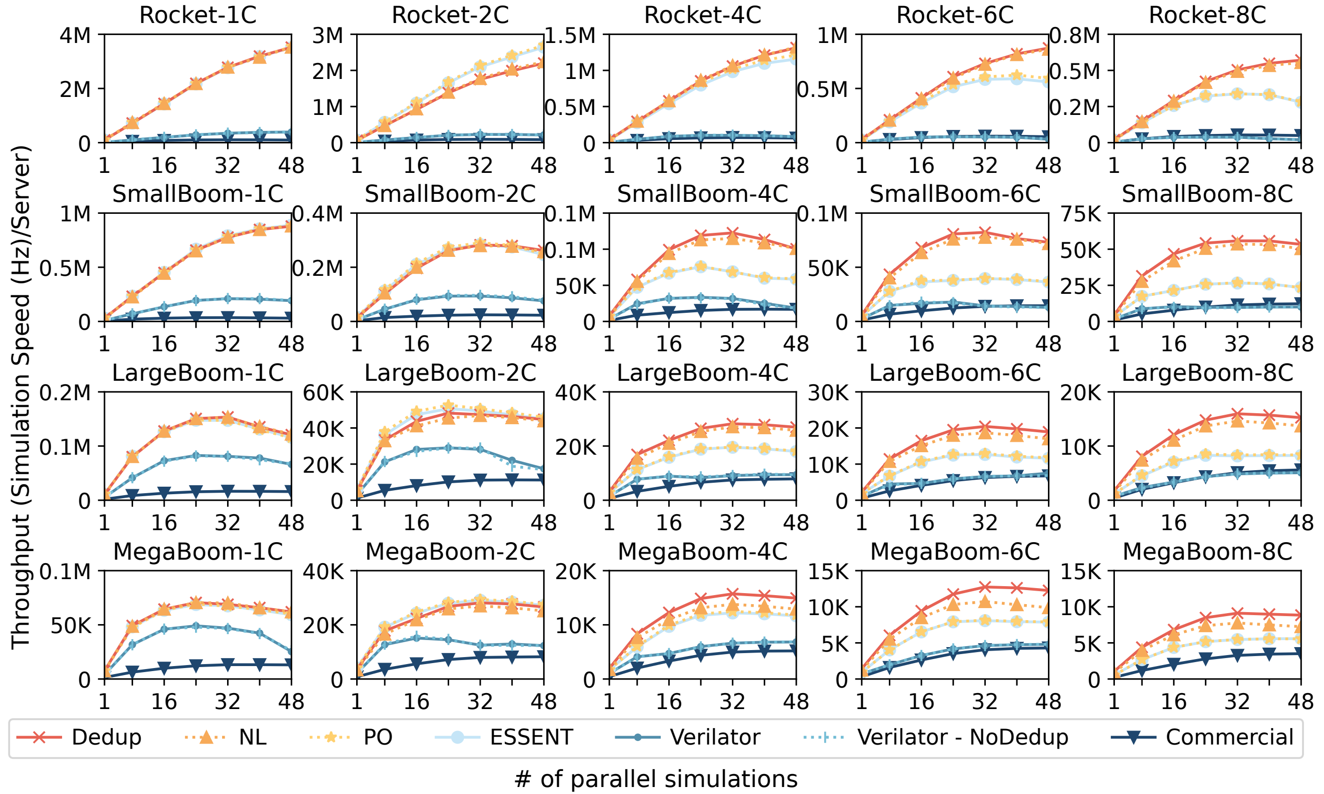


Figure 9. Simulation throughput, measured in aggregate simulation speed per machine, on Server. Larger designs (moving down & right) do not scale as well due to cache contention. Greater amounts of replication (moving right) provide more opportunity for deduplication to benefit. Verilator fails to compile `MegaBoom-8C`. Note different scales on y-axes of subplots.

are scheduled separately. To emulate this behavior, we create multiple copies of the simulator binary, thereby preventing the sharing of code pages by the Linux kernel.

Figure 9 illustrates the measured throughput. Similar to Figure 1, despite the independent execution of each RTL simulator, per-server simulation throughput clearly scales sub-linearly. The enhanced locality and reduced code footprint of `Dedup` significantly elevates per-server simulation throughput compared to the baseline (`ESSENT`). As observed in the single simulation performance (Figure 8), increasing repetition of building blocks in the circuit (e.g. more simulated cores) correlates with more substantial throughput benefits harnessed by `Dedup`. Specifically, on `SmallBoom-8C`, `Dedup` achieves a 2.09× throughput compared to the baseline. As discussed in Section 6.2, the advantages of Verilator's deduplication remain dubious.

Upon closer examination, the minimal throughput difference between `ESSENT` and `PO` once again indicates that the change in partitioning has almost no impact. The significance of our locality-aware scheduling becomes apparent when comparing the throughput of `Dedup` and `NL`.

The throughput of most simulators has diminishing returns after a certain point, indicating the existence of resource contention, in this case for the last-level cache (LLC). As the system approaches this inflection point, there are only marginal gains in throughput to be had from executing more simulations in parallel. Moreover, the significant increase in average simulation time renders higher levels of parallelism less attractive. For example, with the commercial simulator, doubling the number of parallel simulations from 24 to 48 only yields an additional 1.1× throughput, accompanied by an alarming 1.8× increase in completion time (Table 3). The substantial increase of simulation time diminishes the value of the minor throughput benefits.

Table 3. Relative throughput of the commercial simulator, design `SmallBoom-4C`. Because of LLC contention, the increase in throughput is sub-linear, and the average completion time per simulation suffers greatly.

Parallel Simulations	1	8	16	24	32	40	48
Relative Throughput	1.00	5.84	8.28	10.26	11.32	11.45	11.33
Avg. Time (s)	959	1314	1856	2244	2714	3353	4065

We further assess throughput on a platform equipped with additional cache, specifically, AMD's 3D-VCache [25] (Desktop platform in Table 1). While this new technology provides ample LLC and exhibits close to linear scalability for designs of moderate size, the per-server throughput is still constrained by the LLC size on larger designs. Figure 10 shows that Dedup quickly outpaces the other simulators even at lower numbers of parallel simulations, since the smaller simulator binary can fit several times in the available cache, while the other platforms are already competing for the available cache memory again.

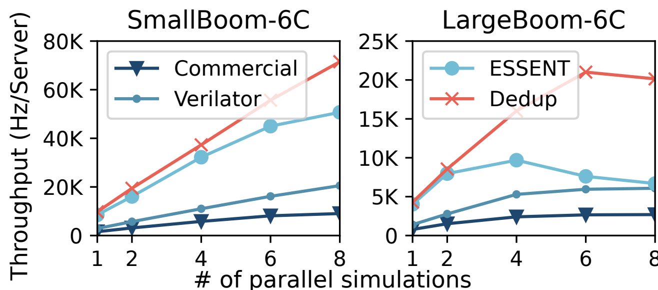


Figure 10. The simulation throughput generally scales better on the cache-rich Desktop platform, but for a large design (right), performance eventually plateaus as different simulations compete for cache resources again.

6.4 Locality-Aware Scheduling Benefits

We analyze the RTL simulators' behavior with hardware performance counters (collected by perf) to understand Dedup's superior performance when cache resources are constrained (Table 4). We restrict the available Last-Level Cache (LLC) for each simulator using Intel RDT to quantify the cache constraint. Intel's cache allocation is implemented with way-based partitioning, which can significantly increase conflict miss rates when cache associativity is relatively low due to partitioning. Despite this limitation, Table 4 still provides valuable insights into the key implications of Dedup.

First, it is expected that Dedup will execute more instructions in total, despite the smaller binary size; for LargeBoom-6C there is a 12.4% increase in instructions executed compared to ESSENT due to indirect memory accesses. However, the dramatical increase in Instructions Per Cycle (IPC) compensates for the deduplication tax. For example, when allocated 4 ways of cache (13 MB), IPC increases from 0.27 to 0.49 (+81%) compared with the baseline (ESSENT). Digging deeper, being able to fit more of the executed code into the L1 instruction cache greatly benefits deduplication, especially considering that the locality-aware scheduling effectively reduces reuse distance. Unlike data cache misses, instruction cache misses are much harder to hide with out-of-order execution.

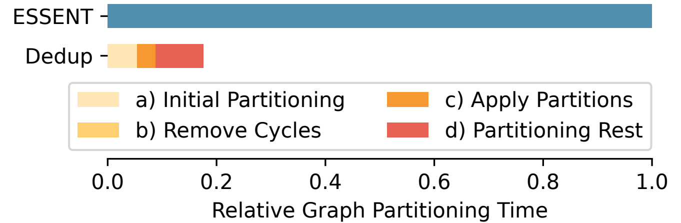


Figure 11. Relative graph partitioning time, normalized to ESSENT, LargeBoom-6C. Stages are described in Figure 7. The Remove Cycles step is too fast to be visible. Partitioning time of Dedup is only 17.6% of ESSENT (blue bar)

Unsurprisingly, limited LLC doesn't constrain private L1 and L2 cache Misses Per Kilo Instructions (MPKI), and larger available LLC benefits LLC MPKI. The drastically reduced branch MPKI is another example of how our locality-aware scheduling reduces reuse distance and thus benefits pipeline components. As such, Dedup leads to fewer pipeline stalls; the performance benefits compensate for the deduplication tax. Once again, performance counters also show that minor differences in the partitioning scheme have a minimal impact (ESSENT compared to P0).

6.5 Deduplication Reduces Graph Partitioning Time

Partitioning the design graph consumes a significant amount of time in ESSENT's flow to create a simulator. Another benefit of deduplication is that it reduces the graph partitioning time since it only needs to partition one instance and can mostly reuse that partitioning for the other instances. The partitioning time benefits of Dedup depend on the input design. As mentioned earlier, Dedup adopts ESSENT's acyclic partitioner and thus should have identical partitioning time if the input design has no replicated modules. However, when partitioning designs with significant reuse, our partitioning scheme achieves a promising speedup. For example, the partitioning time of Dedup is only 17.6% of the ESSENT baseline when partitioning LargeBoom-6C (Figure 11).

6.6 Simulation Duration Impact

In industry, RTL simulations extending beyond a day are not uncommon. However, due to the cache contention effect (Table 3), obtaining results from extended evaluations within a reasonable timeframe proves challenging; for instance, the simulations illustrated in Figure 9 require 25 days in total. In this subsection, we execute much longer simulations for a few data points to assess the impact on the overall results.

The simulation speed of full-cycle simulators like Verilator [20] is independent of the signal activity rate in that simulation. For example, the Verilator simulation of SmallBoom-6C

Table 4. Hardware performance counter measurements on Server for LargeBoom-6C. More cache (moving right) improves most cache metrics, but our optimizations also improve IPC and branch prediction at the expense of more instructions.

Allocated Cache	6.50 MB (2 Ways)				13.00 MB (4 Ways)				19.50 MB (6 Ways)			
	ESSENT	PO	NL	Dedup	ESSENT	PO	NL	Dedup	ESSENT	PO	NL	Dedup
Instructions	1.13 T	1.13 T	1.28 T	1.28 T	1.13 T	1.13 T	1.28 T	1.28 T	1.13 T	1.13 T	1.28 T	1.28 T
Exec Time (s)	1347.39	1369.28	1253.28	960.52	1070.03	1050.37	777.34	679.14	838.52	839.09	611.72	568.31
IPC	0.22	0.21	0.26	0.34	0.27	0.28	0.42	0.49	0.35	0.35	0.54	0.58
L1I MPKI	165.01	164.92	158.82	119.39	162.27	162.75	155.05	118.05	157.50	158.11	152.65	117.12
L1D Read MPKI	19.72	20.26	23.42	27.53	19.69	20.31	23.54	27.51	19.79	20.39	23.55	27.67
L2 MPKI	147.08	147.30	123.75	105.12	147.26	147.85	124.05	105.60	146.83	147.26	124.43	105.66
L3 MPKI	2.67	2.22	2.85	3.11	0.62	0.57	0.67	0.97	0.28	0.25	0.13	0.20
Branch MPKI	19.60	19.87	13.15	12.61	19.54	19.80	13.11	12.59	19.53	19.80	13.12	12.59
Pipeline Stall (%)	84.70	85.17	79.31	73.80	81.11	80.89	67.19	63.48	76.03	76.32	58.58	56.52

maintains an average speed of 2449.05 Hz, with the simulation speed deviating by no more than ±4% from the average 99.29% of the time. This indicates that each signal and logic statement is evaluated consistently, irrespective of its activity level, rendering shorter simulations representative for full-cycle simulators. Conversely, for simulators that account for circuit activity, like ESSENT, simulation performance can vary based on the workload since that can change the amount of activity [4].

To more accurately assess the impact of different workloads and simulation duration, we evaluate SmallBoom-6C using both the original vvadd benchmark, denoted as benchmark A, and an extended version of vvadd, denoted as benchmark B. Benchmark A exhibits an average activity rate of 6.52% on ESSENT, while benchmark B is more dynamic, with an activity rate of 14.87%. Notably, Benchmark B is approximately 11.2× longer in duration than benchmark A.

For this work, we are most concerned with evaluating the benefit of deduplication, i.e. Dedup versus ESSENT. The activity-sensitive simulators (ESSENT & Dedup), execute at approximately half the simulation rate for benchmark B since it has roughly twice the activity. However, Dedup is still significantly faster than ESSENT, which further demonstrates the benefit of deduplication. For a single simulation, deduplication provides a 1.234× speedup on benchmark B, which is marginally higher than on benchmark A (Figure 8). This improvement is attributed to the fact that more active partitions facilitate greater code reuse. For simulation throughput using multiple cores (Figure 12), Dedup realizes a maximum throughput increase of 2.308× over ESSENT using benchmark B, surpassing the 2.079× increase we observe for benchmark A (Figure 9). This enhanced throughput corroborates the hypothesis that increased reuse leads to improved performance. Additionally, the evaluation results do not qualitatively change by running much longer simulations.

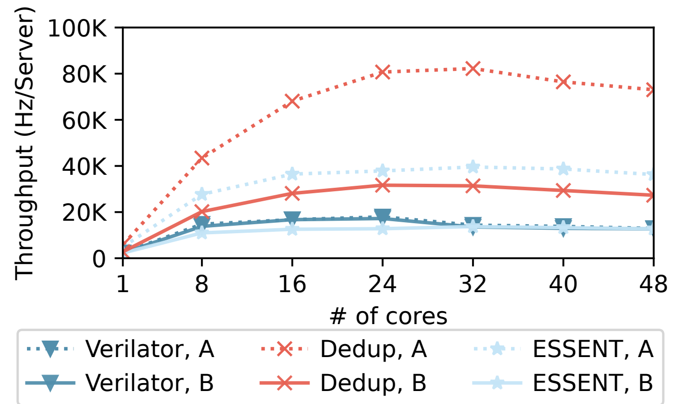


Figure 12. Simulation throughput of SmallBoom-6C, measured in simulation speed per machine on Server platform, varies with different benchmarks, A and B. Dedup achieves a maximum throughput increase of 2.308× compared to ESSENT for benchmark B. An increase in benchmark activity, as in B, results in a decrease in throughput for ESSENT, and similarly for Dedup, which is built upon ESSENT. Different benchmarks have an unnoticeable impact on Verilator's throughput.

7 Related Work

While the burden of RTL simulation throughput in industry data centers is significant, it has not received much attention in academia. Intel [15] reports that throughput scales almost linearly with core count, while other research indicates the existence of peak throughput [21]. Besides, Synopsys [17] reports that extra L3 cache can deliver up to 66% faster performance for a single RTL simulation. Our more systematic experiments support all these conclusions: RTL simulation demands substantial cache, and for small designs that do not exceed the cache size, throughput may scale linearly. However, for large designs, increasing the number of parallel simulations may degrade both throughput and the

average simulation duration due to competition for cache capacity and memory bandwidth.

Recent research in RTL simulation has primarily focused on parallelization techniques to enhance performance. RepCut [22] adopts a software-based approach, significantly reducing inter-thread communication through its novel replication-based partitioning algorithm, and achieves a 27.1× speedup using 24 threads. In the realm of hardware-software co-design, ASH [7] combines a tightly co-designed hardware architecture with a specialized compiler to achieve a 32× speedup compared to the multithreaded Verilator. Manticore [8], on the other hand, eliminates fine-grain synchronization overhead through a static bulk-synchronous parallel execution model, relying solely on the compiler to manage resources and communication, and achieves an 18.0× speedup with an FPGA prototype relative to a single-thread Verilator. Beyond parallelization, Khronos [27] aims to optimize RTL simulation by fusing frequent memory accesses, achieving an average of 2.0× speedup.

The central theme of this work is deduplication, which can synergize with existing approaches to further enhance RTL simulation performance. The deduplication process, by reducing cache requirements and improving data locality, has the potential to increase the efficiency of other RTL simulators. With modest modifications to partitioners and code generation, deduplication could be integrated with these methods, offering additional benefits in terms of performance and throughput.

The granularity of deduplication is worth discussing. Hardware decompilation [19] reads the input netlist, identifies repeated logic in the netlist (such as that which would be synthesized from loops in the original HDL code), and rerolls them into syntactic loops in the recovered HDL code. A mean speedup of 6× was found in RTL simulation. The major barrier to adopting this decompilation work into large-scale RTL simulation is its complexity (e.g. over 20 s to reroll a small shifter). Unlike decompilation, which works at a finer granularity, our approach operates at a coarse granularity, requiring little performance overhead and speeding up graph partitioning time by skipping partitioning of identical instances. Furthermore, our work is not in conflict with hardware decompilation, as deduplication at the module level could provide additional performance benefits when combined with finer-grained decompilation internally.

Apart from academic RTL simulators, deduplication in RTL simulation has been explored within open-source tools, notably Verilator [20] and Arcilator [9]. However, the full potential of deduplication has yet to be realized. These simulators have implemented deduplication on a limited scale, targeting small circuit fragments that bypass cyclic scheduling challenges, which results in modest performance enhancements. Our evaluation (Section 6) demonstrates the constrained benefits of such limited deduplication. In contrast, this work proposes a coarse-grained deduplication strategy. We aim to deduplicate as many circuit partitions as feasible, addressing and resolving the minority that could introduce scheduling cycles. Deduplicating on a larger scale can lead to significant improvements in both performance and throughput.

RTLFlow [16] is a recent work primarily focusing on improving simulation throughput rather than single simulation performance, by exploiting stimuli-level parallelism on a GPU. While it significantly benefits simulation throughput, several drawbacks of RTLFlow limit its application. First, RTLFlow requires over a hundred input stimuli to demonstrate positive throughput benefits, which may not always be available during development. Second, long-tail stragglers will significantly undermine utilization and thus throughput. Finally, running thousands of simulations in parallel requires a considerable amount of GPU memory, and enabling waveform dumping could easily exceed this.

8 Conclusions

In this work, we find limited LLC capacity to be the biggest constraint on RTL simulation throughput. To address this limitation, we propose a coarse-grained deduplication approach for RTL simulation, aiming to enhance both single-simulation performance as well as per-server multi-simulation throughput. The prevalence of replicated components in modern System-on-Chip (SoC) designs makes the exploitation of such duplication highly beneficial for the design iteration cycle.

Avoiding cycles is the primary challenge for circuit deduplication. We address this issue by dissolving partitions that may potentially cause cycles. Our experiments demonstrate that reused module instances can be significantly deduplicated, incurring only minor overheads. Introducing code reuse, we further propose a locality-aware scheduling strategy to effectively reduce reuse distance, thereby benefiting instruction caches and branch predictors. Collectively, our evaluation demonstrates substantial performance benefits for single simulations and more than twofold increase in per-server throughput on designs with high duplication.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful feedback on this work. This material is based upon work supported by, or in part by, the Army Research Laboratory and the Army Research Office under contract/grant W911NF1910466.

A Artifact Appendix

A.1 Abstract

This package contains the source code of ESSENT, our simulator, and Verilator 5.016, along with the necessary software

dependencies to reproduce key results (Figure 2, Figure 8, and Figure 9) of this paper. Figures 1, 10, and 12 are special cases of Figure 9, and thus scripts are not provided.

Due to platform variance (different performance counters on different platforms), we are unable to provide a portable script for collecting data using `perf`.

To quickly verify and reproduce the results in the paper, this package allows users to selectively compile and run simulators. Users can reproduce key results in a few days. Fully reproducing every data point in this paper may take around one month. Please be careful when choosing simulators to test! We recommend testing `simulator_rocket21-1c` and `simulator_boom21-6large`, which complete in a few days (depending on the number of cores).

For more information, please read `README.md`.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** Linux
- **Hardware:** Multi-core x86 platform. If you wish to reproduce Figure 2, a CPU that supports Intel Cache Allocation Technology/Intel Resource Director Technology/AMD Platform Quality of Service Extension, as well as root access, is needed.
- **How much disk space required (approximately)?:** 100 GB
- **How much time is needed to prepare workflow (approximately)?:** A few hours to days, depending on configuration.
- **How much time is needed to complete experiments (approximately)?:** A few days to a month, depending on configuration.
- **Publicly available?:** Yes, artifact available on Zenodo[1] or Docket Hub[2]. Source code available on GitHub[3]
- **Code licenses (if publicly available)?:** BSD
- **Data licenses (if publicly available)?:** BSD
- **Archived (provide DOI)?:** Yes, 10.5281/zenodo.11508626

A.2.1 How to access.

The artifact can be downloaded from Zenodo]. We also have a docker image with the environment properly set up (Click here for docker image).

A.2.2 Hardware dependencies.

A multi-core x86 platform. If you wish to reproduce Figure 2, a CPU that supports Intel Cache Allocation Technology/Intel Resource Director Technology/AMD Platform Quality of Service Extension, as well as root access, is needed.

A.2.3 Software dependencies.

See README.md

A.3 Installation

See README.md

A.4 Experiment workflow

See README.md

[1]Zenodo: https://doi.org/10.5281/zenodo.11508626
[2]Docker Hub: https://hub.docker.com/repository/docker/haooozi/dedup-ae/general
[3]GitHub: https://github.com/ucsc-vama/essent/tree/dedup

A.5 Evaluation and expected results

We recommend selecting a few designs instead of all designs to speed up the evaluation process, as measuring throughput for all simulators may take one month. Our plotting script will plot only available data.

The reproduced Figure 8 should demonstrate the trend that designs with more cores benefit more from deduplication, with minor differences between `Verilator` and `Verilator - NoDedup`.

The reproduced Figure 9 should show the throughput benefits of `Dedup` over the baseline `ESSENT` on designs with more than 2 cores.

A.6 Experiment customization

See README.md

References

[1] Krste Asanović, Rimas Avižienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The Rocket Chip Generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016).

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California) *(DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

[3] Pete Bannon, Ganesh Venkataramanan, Debjit Das Sarma, and Emil Talpes. 2019. Computer and Redundancy Solution for the Full Self-Driving Computer. In *IEEE Hot Chips 31 Symposium (HCS)*. https://doi.org/10.1109/HOTCHIPS.2019.8875645

[4] Scott Beamer. 2020. A Case for Accelerating Software RTL Simulation. *IEEE Micro* 40, 4 (2020), 112–119. https://doi.org/10.1109/MM.2020.2997639

[5] Scott Beamer and David Donofrio. 2020. Efficiently Exploiting Low Activity Factors to Accelerate RTL Simulation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218632

[6] Scott Beamer, Thomas Nijssen, Krishna Pandian, and Kyle Zhang. 2021. ESSENT: A high-performance RTL simuator. In *Workshop on Open-Source EDA Technology (WOSET), at International Conference on Computer-Aided Design (ICCAD)*.

[7] Fares Elsabbagh, Shabnam Sheikhha, Victor A. Ying, Quan M. Nguyen, Joel S Emer, and Daniel Sanchez. 2023. Accelerating RTL Simulation with Hardware-Software Co-Design. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 153–166. https://doi.org/10.1145/3613424.3614257

[8] Mahyar Emami, Sahand Kashani, Keisuke Kamahori, Mohammad Sepehr Pourghannad, Ritik Raj, and James R. Larus. 2024. Manticore: Hardware-Accelerated RTL Simulation with Static Bulk-Synchronous Parallelism. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 219–237. https://doi.org/10.1145/3623278.3624750

[9] Martin Erhart, Fabian Schuiki, Zachary Yedidia, Bea Healy, and Tobias Grosser. 2023. Fast and cycle-accurate hardware simulation in CIRCT. *2023 LLVM Developers' Meeting*. https://llvm.org/devmtg/2023-10/slides/techtalks/Erhart-Arcilator-FastAndCycleAccurateHardwareSimulationInCIRCT.pdf

[10] A. Hemani. 2004. Charting the EDA roadmap. *IEEE Circuits and Devices Magazine* 20, 6 (2004), 5–10. https://doi.org/10.1109/MCD.2004.1364768

[11] Julien Herrmann, M. Yusuf Özkaya, Bora Uçar, Kamer Kaya, and Ümit V. Çatalyürek. 2019. Multilevel Algorithms for Acyclic Partitioning of Directed Acyclic Graphs. *SIAM Journal on Scientific Computing (SISC)* 41, 4 (2019), A2117–A2145. https://doi.org/10.1137/18M1176865

[12] Willy Y. W. Hsu, Jiun-Cheng Tsai, John C. L. Tang, and Charles H. P. Wen. 2020. Profit-Driven Service-Chain Deployment For EDA Requests On Private Cloud. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. 1–4. https://doi.org/10.1109/CloudNet51028.2020.9335794

[13] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 209–216. https://doi.org/10.1109/ICCAD.2017.8203780

[14] Vinaya Kamath, Ravi Giri, and Rajeev Muralidhar. 2013. Experiences with a Private Enterprise Cloud: Providing Fault Tolerance and High Availability for Interactive EDA Applications. In *2013 IEEE Sixth International Conference on Cloud Computing*. 770–777. https://doi.org/10.1109/CLOUD.2013.72

[15] Shesha Krishnapura, Murty Ayyalasomayajula, Shaji Kootaal Achuthan, Vipul Lal, Archana Somasekhara, and Ty Tang. 2022. *IT@Intel: Increasing EDA Performance and Throughput with the Intel® Xeon® Processor Scalable Family*. https://media22.connectedsocialmedia.com/intel/02/19824/IT_Intel_Increasing_EDA_Performance_Throughput_Intel_Xeon_Processor_Scalable_Family.pdf

[16] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. 2023. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In *Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP '22)*. Association for Computing Machinery, New York, NY, USA, Article 88, 12 pages. https://doi.org/10.1145/3545008.3545091

[17] Ramesh Narayanaswamy. 2022. *Boosting EDA Workloads with 3rd Gen AMD EPYC™ Processors*. https://www.synopsys.com/blogs/chip-design/eda-workloads-amd-processors.html

[18] Hasmukh Ranjan. 2011. Cloud computing and EDA: Is cloud technology ready for verification ... (and is verification ready for cloud)?. In *Proceedings of 2011 International Symposium on VLSI Design, Automation and Test*. 1–2. https://doi.org/10.1109/VDAT.2011.5783618

[19] Zachary D. Sisco, Jonathan Balkind, Timothy Sherwood, and Ben Hardekopf. 2023. Loop Rerolling for Hardware Decompilation. *Proc. ACM Program. Lang.* 7, PLDI, Article 123 (jun 2023), 23 pages. https://doi.org/10.1145/3591237

[20] Wilson Snyder. 2017. Verilator: Speedy Reference Models, Direct from RTL. *Presentation to University of Massachusetts Amherst* (2017). https://www.veripool.org/papers/Verilator_Modeling_UMass2017b_pres.pdf

[21] Phil Steinke, Sean Jensen-Grey, Peeyush Tugnawat, and Kari Smayling. 2023. Scaling Synopsys VCS on Google Cloud with AMD Powered VMs. In *Synopsys Users Group (SNUG) Silicon Valley 2023 Proceedings*.

[22] Haoyuan Wang and Scott Beamer. 2023. RepCut: Superlinear Parallel RTL Simulation with Replication-Aided Partitioning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 572–585. https://doi.org/10.1145/3582016.3582034

[23] L.-T. Wang, N. E. Hoover, E. H. Porter, and J. J. Zasio. 1987. SSIM: a software levelized compiled-code simulator. In *Proceedings of the 24th ACM/IEEE Design Automation Conference* (Miami Beach, Florida, USA) *(DAC '87)*. Association for Computing Machinery, New York, NY, USA, 2–8. https://doi.org/10.1145/37888.37889

[24] Zhicheng Wang and Peter M. Maurer. 1991. LECSIM: a levelized event driven compiled logic simulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference* (Orlando, Florida, USA) *(DAC '90)*. Association for Computing Machinery, New York, NY, USA, 491–496. https://doi.org/10.1145/123186.123349

[25] John Wuu, Rahul Agarwal, Michael Ciraula, Carl Dietz, Brett Johnson, Dave Johnson, Russell Schreiber, Raja Swaminathan, Will Walker, and Samuel Naffziger. 2022. 3D V-Cache: the Implementation of a Hybrid-Bonded 64MB Stacked Cache for a 7nm x86-64 CPU. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. 428–429. https://doi.org/10.1109/ISSCC42614.2022.9731565

[26] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanović. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. In *Workshop on Computer Architecture Research with RISC-V (CARRV), International Symposium on Computer Architecture (ISCA)*.

[27] Kexing Zhou, Yun Liang, Yibo Lin, Runsheng Wang, and Ru Huang. 2023. Khronos: Fusing Memory Access for Improved Hardware RTL Simulation. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 180–193. https://doi.org/10.1145/3613424.3614301