When Data-Hash Caching Fails: False Hits in Parameterized Pipeline Search

Thomas Dionysopoulos, CFA

Abstract

Parameterized search—evaluating many pipeline candidates that share sub-computations—is common in hyperparameter optimization, quantitative finance, and experiment management. The natural optimization is sub-expression caching: if two candidates share an intermediate computation, execute it once and reuse the result. We show that the obvious caching strategy—keying on input data hashes—is unsafe. We prove that safe sub-expression caching requires the cache key to induce a *congruence*—an equivalence preserved under composition—on the computation algebra; data-hash keying violates this condition. On a production workload of 100 candidates across a 13-node pipeline, data-hash caching produces 97 false hits: semantically different operations on identical inputs are incorrectly deduplicated. At parameter-sensitive nodes, the false-hit rate reaches 58%. We introduce *identity-gated deduplication*, where cache keys incorporate canonicalized computation identity alongside data references. Identity-gated caching eliminates all false hits while achieving 76.3% CPU reduction at 4.6% overhead. Correctness is verified by 515 comparisons with zero mismatches across 31 independent executions spanning 25 days. We characterize the false-hit failure mode, show that it concentrates at parameter-divergent nodes in the computation DAG, and demonstrate that computation identity is necessary for safe sub-expression caching.

1 Introduction

Many computational workflows evaluate families of related pipelines. A machine learning engineer sweeps hyperparameters; a quantitative researcher tests signal parameters; an experiment platform evaluates model variants. In all cases, candidates share a common structure: an upstream prefix processes shared input data, and parameters diverge at specific nodes downstream.

The waste is substantial. In a production research platform running 100 candidates across 44 sessions, we observed that 1,174 of 1,300 intermediate computations (90.3%) were exact duplicates—the same operation on the same data producing the same output, executed repeatedly because no caching mechanism intervened. In the largest group, log-return computations were executed 69 times; all 69 produced identical output. Sixty-eight were wasted.

The natural optimization is sub-expression caching: store intermediate results and skip re-execution on cache hits. But what should the cache key be?

In build systems such as Make [1] and Bazel [2], the operation is the build rule—it is part of the cache key by construction. In template-generated pipelines, the situation is different: multiple distinct operations are instantiated from a single template definition and share input data. Keying only on input hashes—the DataHash strategy—fails silently.

Table 1 summarizes the result. DataHash caching—keying on the hash of input data flowing into each node—produces 97 false hits, where the cache returns the output of one operation as the result of a different operation that happened to receive the same input. At the signal-computation node, 58% of cache lookups return incorrect results.

IdentityHash caching—incorporating the canonicalized computation identity into the cache key—eliminates all false hits while achieving a higher hit rate (90.5% vs. 83.7%) and greater compute reduction (89.8% vs. 75.5%). The measured execution gate achieves 76.3% CPU reduction at 4.6% overhead, verified correct by 515 comparisons with zero mismatches.

This paper makes three contributions:

Table 1: Three caching strategies on a production workload (100 candidates, 1,300 node evaluations, 44 sessions). DataHash achieves high hit rate but produces 97 false hits. IdentityHash achieves a higher hit rate with zero false hits.

Strategy	Exec	Skip	Hit%	False	Safe
No cache	1,300	0	0.0	0	✓
DataHash	212	1,088	83.7	**97**	×
IdentityHash	123	1,177	90.5	0	✓

1. We identify and characterize the *false-hit* failure mode in sub-expression caching for parameterized pipelines: data-hash caching silently returns incorrect results when different operations share inputs (section 4).
2. We show that false hits concentrate at *parameter-divergent nodes* in the computation DAG, with rates determined by the parameter topology—58% at the signal node, 31% at the risk node (section 5).
3. We introduce *identity-gated deduplication*, implement it as an execution gate, and measure 76.3% CPU reduction with zero false hits, verified by 515 comparisons across 31 independent runs (section 6).
4. We prove that safe compositional caching requires the cache key to induce a *congruence* on the computation algebra (section 5.4). Data-hash keying fails this algebraic condition; identity-gated keying satisfies it.

2 Background and Related Work

2.1 Build Systems and Content-Addressed Caching

Content-addressed caching is well-established in build systems. Make [1] introduced dependency-based rebuilding. Bazel [2] hashes build rules and their inputs to produce cache keys. Nix [3] uses content-addressed derivations for reproducible builds. Mokhov et al. [4] provide a unifying framework for build system design.

These systems cache at *rule* or *derivation* granularity: each cache entry corresponds to a named build target. Crucially, the operation (build rule) is always part of the key. The false-hit problem does not arise because different rules have different names. In parameterized pipelines, however, operations are generated from templates at runtime and do not have pre-assigned names. The system must derive the operation identity from the expression, not from a static rule definition.

2.2 Hyperparameter Search and Experiment Management

Hyperparameter optimization frameworks—Optuna [5], Ray Tune [6]—evaluate many model configurations over shared training data. Each trial is an independent execution; sub-expression sharing across trials is not exploited. MLflow [7] tracks experiment metadata but does not deduplicate sub-computations. These systems face the same structural opportunity: trials that share preprocessing could reuse intermediate results. None addresses the cache-key safety problem we characterize.

2.3 Incremental and Differential Computation

Adapton [8] provides demand-driven incremental computation. Pugh and Teitelbaum [9] introduce incremental attribute evaluation. Differential dataflow [10] supports incremental computation with data sharing. Self-adjusting computation [11] maintains outputs under input changes. Spark's RDD lineage [12] enables recomputation from stable intermediate representations.

These systems address *re-execution after input changes*. Our problem is different: inputs do not change between candidates. The *computation itself* varies while sharing sub-computations. The challenge is detecting which sub-computations are shared and which differ—and ensuring that the caching mechanism does not confuse them.

2.4 Memoization

Function-level memoization caches return values by argument identity. Three differences distinguish our setting:

1. In our system, `dlog` and `log_ret` are aliases for the same operation. Identity-gated caching canonicalizes before hashing; standard memoization does not.
2. Memoization caches at function-call granularity. Identity-gated caching operates at sub-expression granularity within a DAG, sharing intermediate nodes across pipeline instances.
3. Memoization keys on input values. For parameterized pipelines, this produces false hits (identical inputs, different operations). Adding the operation to the key fixes the problem—but requires canonicalization and templatization machinery that standard memoization does not provide.

More fundamentally, function-level memoization requires only per-call soundness: $k(x) = k(y) \Rightarrow f(x) = f(y)$ [9]. Compositional pipeline caching requires a stronger condition: the key equivalence must be a *congruence*—preserved under composition with downstream operations (section 5.4). This distinction has not been identified in the memoization literature.

3 System Model

3.1 Parameterized Pipeline DAG

A *pipeline template* T defines a directed acyclic graph of n nodes $\{v_1, \ldots, v_n\}$, where each node v_i applies an operation f_i to the output of its predecessors. A *parameter vector* θ instantiates the template: operations at parameter-sensitive nodes are specialized by θ.

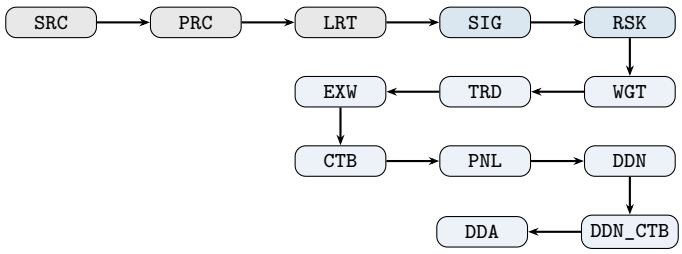
Definition 1 (Parameter-divergent node)**.** *A node v_i in template T is* parameter-divergent *if there exist parameter vectors $\theta_a \neq \theta_b$ such that $f_i^{\theta_a} \neq f_i^{\theta_b}$.*

Definition 2 (Shared prefix)**.** *The* shared prefix *of template T is the maximal set of nodes $S \subseteq \{v_1, \ldots, v_n\}$ such that for all $v_i \in S$ and all parameter vectors θ_a, θ_b: $f_i^{\theta_a} = f_i^{\theta_b}$.*

A *parameterized search* evaluates T on k parameter vectors $\{\theta_1, \ldots, \theta_k\}$ against shared input data D. This produces $k \times n$ node evaluations, of which only the unique (operation, input) pairs require execution.

3.2 The Experimental Pipeline

Our study uses a 13-node quantitative finance pipeline:



Gray: shared prefix (parameter-independent). Blue: parameter-divergent. Light blue: downstream of divergence.

The template is parameterized by two values: a signal window (`SIG_WIN`) and a risk window (`RSK_WIN`). Nodes SRC, PRC, and LRT form the shared prefix—they apply the same operations regardless of parameters. SIG is the first parameter-divergent node (sensitive to `SIG_WIN`). RSK and all downstream nodes depend on both parameters.

This pipeline structure is representative of parameterized search in general: an upstream data-preparation prefix feeds into parameter-sensitive computation. The same structure appears in ML preprocessing-then-training pipelines, ETL-then-analysis workflows, and multi-model experiment sweeps.

3.3 Identity Construction

The pipeline and its caching mechanism are implemented in Blisp, a domain-specific language for quantitative finance with a six-stage execution pipeline:

1. **Parse**: source expression → AST.
2. **Normalize**: alias resolution (`dlog` → `log_ret`, `w5` → `wkd`).
3. **Canonicalize**: structural normalization of the AST.
4. **Plan**: IR node assignment.
5. **Optimize**: fusion and rewriting.
6. **Execute**: evaluation.

The computation hash is computed after canonicalization:

$$\texttt{MOR\_HSH} = \text{SHA256}(\textit{canon\_expr} \parallel \textit{prm\_ser} \parallel \texttt{DIC\_HSH}) \tag{1}$$

where *canon_expr* is the canonicalized expression, *prm_ser* is a deterministic serialization of the parameter vector, and `DIC_HSH` is a SHA-256 fingerprint of the runtime operation catalogue (ensuring that cache entries are invalidated when the set of available operations changes).

Canonicalization ensures that different spellings of the same operation produce the same hash. `dlog`, `log_ret`, and `(-> (log) (diff 1))` all resolve to the same canonical form and the same `MOR_HSH`.

Data source paths are excluded from `MOR_HSH`: two pipelines with identical operations but different data sources share the same computation hash. The executable identity adds data and environment:

$$\texttt{SRH\_HSH} = \text{SHA256}(\texttt{MOR\_HSH} \parallel \texttt{ENV\_HSH} \parallel \text{sort}(\textit{DAT})) \tag{2}$$

4 The False-Hit Problem

Definition 3 (False hit)**.** *A* false hit *occurs when a cache lookup returns a result r_a for a query q_b where r_a is the correct result for a different computation q_a, and $r_a \neq r_b$ (the correct result for q_b).*

False hits are distinct from cache misses and from hash collisions. A false hit occurs when the cache key *correctly identifies the inputs* but *fails to distinguish the computation.*

4.1 Mechanism

Consider two candidates c_a and c_b with parameter vectors $\theta_a = (\texttt{SIG\_WIN}{=}25)$ and $\theta_b = (\texttt{SIG\_WIN}{=}50)$. At node SIG, both candidates receive the same input (the log-return series from node LRT).

Under DataHash caching:

$$\text{key}(c_a, \text{SIG}) = \text{SHA256}(\text{output}(\text{LRT}))$$
$$\text{key}(c_b, \text{SIG}) = \text{SHA256}(\text{output}(\text{LRT}))$$

The keys are identical. If c_a executes first, the cache stores $\text{SIG}(25, \text{LRT})$. When c_b queries the cache, it receives $\text{SIG}(25, \text{LRT})$ instead of $\text{SIG}(50, \text{LRT})$. This is a false hit: a silently incorrect result with no error signal.

Under IdentityHash caching:

$$\text{key}(c_a, \text{SIG}) = \text{SHA256}(\texttt{MOR\_HSH}_a \parallel \text{output}(\text{LRT}))$$
$$\text{key}(c_b, \text{SIG}) = \text{SHA256}(\texttt{MOR\_HSH}_b \parallel \text{output}(\text{LRT}))$$

Since $\theta_a \neq \theta_b$, we have $\texttt{MOR\_HSH}_a \neq \texttt{MOR\_HSH}_b$, and the keys differ. No false hit occurs.

Table 2: Per-node false-hit analysis. DataHash produces 97 false hits concentrated at parameter-divergent nodes (SIG: 58%, RSK: 31%). IdentityHash produces zero false hits at every node while achieving equal or higher hit rates.

Node	Total	DataHash hits	DataHash false	DataHash false%	IdentityHash hits	Node type
SRC	100	97	0	0%	97	shared
PRC	100	97	0	0%	100	shared
LRT	100	97	0	0%	97	shared
SIG	100	39	**58**	**58%**	92	*divergent*
RSK	100	66	**31**	**31%**	95	*divergent*
WGT	100	90	0	0%	90	downstream
TRD	100	90	0	0%	90	downstream
EXW	100	82	**8**	**8%**	86	downstream
CTB	100	86	0	0%	86	downstream
PNL	100	86	0	0%	86	downstream
DDN	100	86	0	0%	86	downstream
DDN_CTB	100	86	0	0%	86	downstream
DDA	100	86	0	0%	86	downstream
Total	**1,300**	**1,088**	**97**	**7.5%**	**1,177**	

5 Structural Analysis of False Hits

5.1 Concentration at Parameter-Divergent Nodes

False hits are not uniformly distributed across the DAG. They concentrate at parameter-divergent nodes—precisely the nodes where different operations act on shared inputs.

Table 2 shows the distribution. All 97 false hits under DataHash occur at three nodes: SIG (58), RSK (31), and EXW (8). These are the nodes where the parameter vector changes the operation while the input data remains identical.

Shared-prefix nodes (SRC, PRC, LRT) produce no false hits under either strategy: the operation is the same for all candidates, so identical inputs do imply identical outputs.

Downstream nodes (WGT through DDA) mostly produce no false hits under DataHash because their inputs have already diverged—the output of a parameter-divergent ancestor differs across candidates, producing different input hashes. EXW is the exception: 8 false hits arise from parameter combinations that produce identical intermediate results at an upstream node despite differing parameters.

5.2 False-Hit Rate and Parameter Diversity

The false-hit rate at a parameter-divergent node depends on the ratio of unique parameter values to total candidates. At SIG, with 100 candidates and 8 unique `SIG_WIN` values: under DataHash, only the first candidate with each unique parameter value executes correctly. The remaining candidates receive false hits from candidates with different parameters but the same input.

Property 1 (False-hit rate)**.** *At a parameter-divergent node with u unique operations and k total candidates sharing the same input data, DataHash produces at most $k - u$ false hits. The false-hit rate approaches $1 - u/k$ as k grows.*

On our workload: SIG has u=8 unique operations across k=100 candidates with the same data, yielding a predicted upper bound of $100 - 8 = 92$ false hits. The observed count is 58, lower than the bound because not all 100 candidates share the same source data (3 distinct data sources exist in the workload).

5.3 Structural Implications

The false-hit rate is a structural property of the parameter topology, not a property of the dataset, the implementation, or the workload volume. *Any* caching strategy that keys only on input data will produce false hits in proportion to the number of distinct operations applied to shared inputs. The only way to eliminate false hits without sacrificing deduplication is to incorporate the operation into the cache key.

5.4 Formal Characterization

The false-hit phenomenon admits a precise algebraic characterization. In universal algebra, a *congruence* on an algebra $(A, \circ)$ is an equivalence relation $\sim$ that is preserved by all operations: $a_1 \sim a_2$ implies $C[a_1] \sim C[a_2]$ for every context $C[\cdot]$ [16]. The quotient $A/\sim$ inherits a well-defined algebra structure only when $\sim$ is a congruence.

Definition 4 (Compositional cache soundness)**.** *Let $(A, \circ)$ be an algebra of pipeline computations and* $\text{eval} : A \to R$ *the evaluation function. A cache key function $k : A \to K$ is* compositionally sound *if (i)* $\ker(k) \subseteq \ker(\text{eval})$ *(no false hits at individual nodes) and (ii)* $\ker(k)$ *is a congruence on $(A, \circ)$ (no false hits propagate through composition).*

Theorem 1 (Cache key correctness)**.** *A cache key function is safe for compositional reuse if and only if the equivalence relation it induces is a congruence on the computation algebra that refines the evaluation equivalence.*

Proof. ($\Rightarrow$) Suppose $\ker(k)$ is not a congruence. Then there exist a_1, a_2 with $k(a_1) = k(a_2)$ but $k(C[a_1]) \neq k(C[a_2])$ for some context C. The cache deduplicates a_1 and a_2, storing a single result. When C consumes this result, $\text{eval}(C[a_1]) \neq \text{eval}(C[a_2])$: the downstream context receives the wrong intermediate, producing a silent error.

($\Leftarrow$) If $\ker(k) \subseteq \ker(\text{eval})$ and $\ker(k)$ is a congruence, then $k(a_1) = k(a_2)$ implies both $\text{eval}(a_1) = \text{eval}(a_2)$ (local soundness) and $k(C[a_1]) = k(C[a_2])$ for all C (compositional soundness). Cached results are correct at every node and remain correct through downstream composition. □

Corollary 1 (DataHash violates the congruence condition)**.** *DataHash induces $a_1 \sim a_2$ when the input data is identical. At parameter-divergent nodes,* $\text{SIG}(25, d) \sim \text{SIG}(50, d)$ *(same input d), but their outputs differ and downstream contexts distinguish them:* $\text{RSK}(\text{SIG}(25, d)) \not\sim \text{RSK}(\text{SIG}(50, d))$*. The 97 false hits in table 2 are empirical evidence of this congruence failure.*

Corollary 2 (IdentityHash satisfies the congruence condition)**.** *IdentityHash induces $a_1 \sim a_2$ when the canonicalized computation identity and input data are both identical. Canonicalization is context-free (each operation name is resolved independently in the AST), so canonical equivalence is preserved under composition: if* $\text{canon}(a_1) = \text{canon}(a_2)$*, then* $\text{canon}(C[a_1]) = \text{canon}(C[a_2])$ *for all contexts C. Therefore* $\ker(\textit{IdentityHash})$ *is a congruence, and identity-gated caching is compositionally sound.*

The three rows of table 1 correspond to three algebraic regimes: no equivalence (execute everything), a non-congruence (produces false hits at composition boundaries), and a congruence (safe reuse with zero false hits).

6 Identity-Gated Deduplication

6.1 Execution Gate

The execution gate is a lookup in an in-memory hash map keyed on the SHA-256 hash of the serialized identity tuple:

```
for each candidate c in sweep:
  key = SHA256(serialize(c.identity_tuple))
  if cache[key] exists:
    result = cache[key]
```

Table 3: Execution gate measurements. Baseline: 8 unique candidates, single sweep. Multi-sweep: same candidates repeated in the same process. Large: 18 unique candidates, two sweeps.

Configuration	Exec	Hits	Time (ms)	Reduction
Baseline (1×8)	8	0	3,823	—
Two sweeps (2×8)	8	8	907	76.3%
Three sweeps (3×8)	8	16	964	91.6%
Large (2×18)	18	18	5,269	52.0%

```
else:
  result = execute(c)
  cache[key] = result
```

The identity tuple includes the computation hash (`MOR_HSH`), parameter values, data references, and the runtime catalogue fingerprint (`DIC_HSH`). Two candidates with different parameters produce different identity tuples and different cache keys, even when their input data is identical.

6.2 Measured Performance

The execution gate was implemented in the production system and measured on real workloads.

Table 3 shows the results:

- **CPU reduction**: 76.3% on the two-sweep configuration. The second sweep of 8 candidates completes in 907 ms (cache lookup only) vs. the 3,823 ms baseline.
- **Wall-clock reduction**: 74.8% on the same configuration.
- **Cache overhead**: 4.6% of baseline execution time. SHA-256 computation and hash-map lookup are negligible relative to node execution (per-node times range from 0.5 to 10.4 ms).
- **Scaling**: Three sweeps achieve 91.6% reduction. Amortization improves with sweep count because cache population occurs only on the first sweep.
- **Hit rate**: 100% on repeated sweeps (all candidates hit on every subsequent sweep).

The per-node compute times in this workload (0.5–10.4 ms) reflect a small dataset (6 columns, ∼10K rows). On production-scale data (500+ columns, 20+ years of daily observations), per-node execution times scale to seconds or minutes. The gate's overhead—hash computation and hash-map lookup—is constant regardless of data size, so the savings ratio improves with scale.

6.3 Correctness Verification

Correctness was verified at three levels:

1. Cross-run determinism. 31 independent runs spanning 25 days (2026-05-09 to 2026-06-03) were grouped into 4 determinism groups by parameter vector and source data hash. Within each group, all 13 intermediate node hashes were compared across all member runs. Result: 403 comparisons, 0 mismatches.

2. Cross-process file comparison. The same 8-candidate sweep was executed in two separate processes with no shared memory. All 104 node-level CSV output files were compared byte-for-byte. Result: bit-identical.

3. Within-process cache verification. Results returned from cache hits were compared against results from fresh execution in the same process via content hashes. All 8 comparisons matched.

Table 4: Correctness evidence. 515 total comparisons, zero mismatches. Outputs are bit-identical, not approximately equal.

Test	Compared	Mismatches	Scope
Cross-run determinism	403	0	31 runs, 25 days, 4 groups
Cross-process files	104	0	2 processes, byte-identical
Within-process hashes	8	0	cache hit vs. fresh execution
Total	**515**	**0**	

7 Discussion

7.1 When Is Identity-Gated Caching Necessary?

Identity-gated caching is necessary whenever a system evaluates multiple computations that share input data but differ in their operations. This occurs in:

- **Hyperparameter sweeps**: grid search or random search over model parameters, where data preprocessing is shared across trials.
- **Strategy backtesting**: evaluating financial strategies with different signal or risk parameters on shared market data.
- **A/B testing pipelines**: comparing variants that differ at one stage while sharing upstream data preparation.
- **Multi-model evaluation**: comparing model architectures that share feature extraction.

In all these cases, data-hash caching is unsafe whenever parameter-divergent nodes exist.

7.2 The "Just Add the Function Name" Objection

A natural objection is that the false-hit problem is solved by adding the function name to the cache key. This is correct in principle, and computation identity does exactly this. Three aspects make the realization non-trivial:

1. **Canonicalization.** The "function name" must be canonical. In our system, `dlog`, `log_ret`, and the composed expression `(-> (log) (diff 1))` are different spellings of the same operation. A six-stage canonicalization pipeline resolves all aliases to a single canonical form before hashing.
2. **Templatization.** The data reference must be separable from the computation. Without explicit separation, `file("a.csv")` and `file("b.csv")` produce different keys even when the computation is identical. `MOR_HSH` excludes data paths, enabling deduplication of computation templates independently of data binding.
3. **Catalogue versioning.** The cache key includes a fingerprint of the runtime operation catalogue (`DIC_HSH`), ensuring that entries are automatically invalidated when the operation semantics change across runtime versions.

7.3 Scope and Limitations

The false-hit mechanism is topology-independent: it depends only on the existence of parameter-divergent nodes receiving shared inputs. Every parameterized pipeline template has parameter-divergent nodes by definition. The *rate* of false hits depends on the specific parameter topology—the number of unique parameter values and the data-source diversity—and will differ across templates.

The measured compute reduction (76.3%) is specific to this workload. The reduction for a given template approaches $1 - u/t$, where u is the number of unique sub-computations and t is the total evaluations. Workloads with more candidates and fewer unique parameter configurations will see higher reductions.

8 Threats to Validity

Single template. All experiments use one pipeline template (13 nodes, 2 parameters). The false-hit mechanism is topology-independent, but the measured rates and reduction ratios are template-specific. Templates with different topologies (branching rather than linear, more parameter-divergent nodes) would produce different false-hit distributions.

In-memory cache. The current implementation uses an in-memory hash map, cleared on process exit. Persistent caching introduces additional concerns: data staleness (the input file may change without the path changing), serialization overhead, and storage management. The data axis currently uses file paths, not content hashes.

Workload characteristics. The 90.3% duplication rate reflects a research workflow with many repeated runs. Production workloads with less repetition will show lower duplication. The false-hit *rate* (58% at the signal node) is a property of the parameter topology, not the workload volume.

No energy measurement. We report CPU time and wall-clock time, not joules or carbon. CPU reduction is a necessary precondition for energy reduction but is not equivalent to it. We do not claim energy savings.

Single architecture. Determinism is verified on one hardware architecture (x86-64) and one runtime version. The catalogue hash (`DIC_HSH`) changes across versions by design, preventing cross-version cache reuse. Cross-architecture floating point determinism is not tested.

9 Conclusion

Data-hash caching in parameterized pipelines is unsafe. On a production workload, it produces a 7.5% false-hit rate overall and 58% at the most parameter-sensitive node—silently returning incorrect results with no error signal.

We characterized this failure mode, showed that it concentrates at parameter-divergent nodes in the computation DAG, and demonstrated that identity-gated deduplication—incorporating canonicalized computation identity into cache keys—eliminates all false hits ($97 \to 0$) while achieving 76.3% CPU reduction at 4.6% overhead. Correctness is verified by 515 comparisons with zero mismatches across 31 runs spanning 25 days, with bit-identical outputs.

The false-hit problem is structural: it arises whenever template-generated operations share inputs, regardless of the specific template or domain. The failure is algebraic: safe compositional caching requires the cache key to induce a congruence on the computation algebra (theorem 1), a condition that data-hash keying violates and identity-gated keying satisfies. Identity-gated caching is the mechanism that makes sub-expression deduplication safe.

References

[1] S. I. Feldman. Make—a program for maintaining computer programs. *Software: Practice and Experience*, 9(4), 1979.

[2] Google. Bazel: a fast, scalable, multi-language build system. `https://bazel.build`, 2015.

[3] E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In *LISA*, 2004.

[4] A. Mokhov, N. Mitchell, and S. Peyton Jones. Build systems à la carte. *Proc. ACM Program. Lang.*, 2(ICFP), 2018.

[5] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework. In *KDD*, 2019.

[6] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. Tune: A research platform for distributed model selection and training. In *ICML AutoML Workshop*, 2018.

[7] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, F. Xie, and C. Zuber. Accelerating the machine learning lifecycle with MLflow. *IEEE Data Eng. Bull.*, 41(4), 2018.

[8] M. A. Hammer, K. U. Phang, M. Hicks, and J. S. Foster. Adapton: Composable, demand-driven incremental computation. In *PLDI*, 2014.

[9] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *POPL*, 1989.

[10] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*, 2013.

[11] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Carnegie Mellon University, 2005.

[12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[13] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame. Nextflow enables reproducible computational workflows. *Nature Biotechnology*, 35(4), 2017.

[14] F. Mölder *et al.* Sustainable data analysis with Snakemake. *F1000Research*, 10(33), 2021.

[15] H. Esfahani, J. Fietz, Q. Ke, A. Kolber, E. Lan, E. Mavrinac, W. Schulte, N. Sanber, and S. Sezgin. CloudBuild: Microsoft's distributed and caching build service. In *ICSE Companion*, 2016.

[16] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon, 1970.