Fingerprint Indexing for Paramodulation and Rewriting

(Extended Version)

Stephan Schulz

Institut für Informatik, Technische Universität München,
D-80290 München, Germany, `schulz@eprover.org`

Abstract. Indexing is critical for the performance of automated first-order theorem provers. We introduce *fingerprint indexing*, a non-perfect indexing technique that is based on short, constant length vectors of samples of term positions ("fingerprints") organized in a trie. Fingerprint indexing supports both matching and unification as retrieval relations. The algorithms are simple, the indices are small, and performance is very good in practice.
We have implemented fingerprint indexing in the equational theorem prover E, where it is used for backwards rewriting and superposition. We demonstrate the performance of the index both in relative and absolute terms using large-scale profiling.

1 Introduction

Saturating theorem provers based on resolution or superposition are among the most powerful ATP systems for first-order logic. Well-known examples include Vampire [6], SPASS [14] and E [8, 10]. These systems work in a refutational setting. The state of the proof search is represented by a set of clauses. It is manipulated using inference rules, with generating rules adding new clauses, and simplification rules removing or modifying clauses. The proof search ends successfully if the empty clause can be derived, making the contradiction between axioms and negated hypothesis explicit.

While some rules operate on individual clauses, the most important inferences (resolution and paramodulation/superposition) and simplifications (rewriting and subsumption) use two or more premises—usually a main premise and one or more additional side premises. For completeness, it is critical that *all* non-redundant inferences are eventually performed. Similarly, search performance is usually much improved if as many simplifications as possible take place. This requires the system to find potential inference partners for a given clause in the potentially large set of clauses representing the search space.

Many early systems relied on simple sequential search through all possibilities for this. However, it soon became obvious that this approach is very expensive for large proof states. The performance of deduction systems can be improved if this sequential search is replaced by a more efficient mechanism, namely *indexing* of potential inference partners. An index, in this context, is a data structure with associated algorithms that allows the efficient retrieval of terms or clauses that are in a given *retrieval relation* with a query from the indexed set of clauses. Overviews of many indexing techniques can be found in [2] and [11].

Most early proof problems for theorem provers have been carefully handcrafted, and contain no extraneous symbols or axioms. However, this situation has changed profoundly over time. Recent releases of the TPTP problem library [13] contain many real application problems that include thousands to millions of clauses and formulas, either from a large, general knowledge base, or as part of an automatically generated problem. While very few of these axioms are needed for proving any given conjecture, they all are part of the proof state. As a result, the number of terms has grown, but the likelihood of any two terms being suitable for an inference has dropped. Thus, efficient indexing has become even more important for performance on relevant problems in recent years.

E has featured *(perfect) discrimination tree indexing* [4] for forward unit rewriting since version 0.1, published in 1998. It added feature vector indexing for subsumption and subsumption resolution (called *contextual literal cutting* in E's equational setting) in version 0.8. However, until recently it did not implement any indexing for backward simplification (rewriting of processed clauses by newly derived unit clauses) or the main generating inference (superposition, a version of paramodulation restricted by term orderings).

While indexing is concerned with finding potential partners among clauses, most indexing techniques index (first-order) *terms*, and are then lifted to (occurrences of terms in) clauses. One exception to this is *feature vector indexing* [9], a technique for finding candidate clauses for subsumption. Feature vector indexing is a non-perfect technique (i.e. not every returned candidate will actually subsume or be subsumed by the query), but it has a very simple structure, easy and efficiently implementable algorithms, and shows good performance in practice.

In this paper, we present *fingerprint indexing*, a new term indexing technique that can be seen as a generalization of top symbol hashing. It shares the basic structure of feature vector indexing (indexed objects are represented by finite-length vectors organized in a trie), and combines it with ideas from coordinate and path indexing [12, 4, 1] (values in the index vectors represent the occurrence of symbols at certain positions in terms). The resulting index can be used for retrieving candidate terms unifiable with a query term, matching a query term, or being matched by a query term. The index data structure has very low memory use, and all operations for maintenance and candidate retrieval are fast in practice. Variants of fingerprint indexing have been incorporated into E for backwards simplification and superposition, with generally positive results.

The remainder of this paper describes the ideas behind, and performance of, fingerprint indexing. We begin by introducing the necessary notions of first-order logic with equality. In the next section, we define fingerprint samples, fingerprints, and fingerprint indexing.

Section 5 demonstrates the performance of fingerprint indexing by analyzing data from thousands of test problems solved by a version of E instrumented to allow large scale profiling of the indexing and general superposition algorithms, comparing fingerprint indexing and non-perfect discrimination tree indexing. We conclude with a summary and some notes on future work.

2 Terminology

We use standard terminology for first-order clausal logic. A signature consists of a finite set F of function symbols with associated arities. We write $f|_n$ to indicate that f has arity $n \in \mathbb{N}_0$. We use a, b, c, d, e for constants (elements of F with arity 0) and f, g, h, j for non-constant function symbols, possibly with subscripted indices. We assume an enumerable set V of *variables* disjoint from F, typically denoted by $x, y, z, x_0, \ldots$, or by upper-case `X0`,`X1` if represented in TPTP syntax. The set $\mathit{Term}(F, V)$ of *terms* over F and V is the smallest set such that (i) $x \in \mathit{Term}(F, V)$ for all $x \in V$ and (ii) if $f|_n \in F$ and $t_1, \ldots, t_n \in \mathit{Term}(F, V)$, then $f(t_1, \ldots, t_n) \in \mathit{Term}(F, V)$. In that case, the t_i are called the *direct subterms* of $f(t_1, \ldots, t_n)$. A *subterm* of $t \in \mathit{Term}(F, V)$ is either t itself, or, recursively, a subterm of a direct subterm of t. A *proper subterm* of t is a subterm t' of t that is different from t. We usually omit the parentheses from constant function symbols, as e.g. in $f(g(x), a)$.

An (equational) *atom*[1] is an unordered pair of terms, written as $s \simeq t$. A *literal* is either an atom, or a negated atom, written as $s \not\simeq t$. If we want to write about arbitrary literals without specifying polarity, we use $s \dot{\simeq} t$, or, in less precise way, $l, l_1, l_2, \ldots$. Note that $\simeq$ is symmetric in this notation.

A *clause* is a multi-set of literals, interpreted as an implicitly universally quantified disjunction, and usually written as $l_1 \vee l_2 \ldots \vee l_n$. Please note that in this notation, the $\vee$ operator is associative and commutative. The empty clause is written as $\square$, and the set of all clauses as $\mathit{Clauses}(F, V)$. A *formula* in clause normal form is a multi-set of clauses, interpreted as a conjunction.

A *substitution* is a mapping $\sigma : V \rightarrow \mathit{Term}(F, V)$ with the property that $\mathit{Dom}(\sigma) = \{x \in V \mid \sigma(x) \neq x\}$ is finite. It is extended to a function on terms, atoms, literals and clauses in the obvious way. If σ and τ are two substitutions, σ is called *more general* than τ, if there exists a substitution σ' such that $\tau = \sigma \circ \sigma'$. It is called *strictly more general*, if σ is more general than τ, but not vice versa.

A *matcher* from a term s to another term t is a substitution σ such that $\sigma(s) \equiv t$. A *unifier* of two terms s and t a substitution σ such that $\sigma(s) \equiv \sigma(t)$. If s and t are unifiable, a *most general unifier (mgu)* for them exists, and is unique up to variable renaming.

We use a slightly expanded definitions for positions in terms, allowing positions that do not lie in a term: A *(potential) position* is a sequence $p \in \mathbb{N}^*$ over natural numbers. We write a position of length (or depth) n as $p = i_1.i_2 \ldots i_n$, and use ϵ to denote the empty position. The set of positions in a term, $\mathrm{pos}(t)$ is defined as follows: If $t \equiv x \in V$, then $\mathrm{pos}(t) = \{\epsilon\}$. Otherwise $t \equiv f(t_1, \ldots, t_n)$.

[1] For our current discussion, the non-equational case is a simple special case and can be handled by encoding non-equational atoms as equalities with a reserved constant $\$true$.

In this case $\text{pos}(t) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n, p \in \text{pos}(t_i)\}$. The subterm of t at position $p \in \text{pos}(t)$ is defined recursively: if $p = \epsilon$, then $t|_p = t$. Otherwise, $p \equiv i.p'$ and $t \equiv f(t_1, \ldots, t_n)$. In that case, $t_p = t_i|_{p'}$. The top symbol of $x \in V$ is $\text{top}(x) = x$ and the top symbol of $f(t_1, \ldots, t_n)$ is $\text{top}(f(t_1, \ldots, t_n)) = f$. We write $s[p \leftarrow t]$ to denote the term constructed from s by replacing the subterm $s|_p$ with t.

Positions can be extended to literals (selecting a term in a literal) and clauses (selecting a term in a literal in a clause) easily if we assume an arbitrary, but fixed ordering of terms in literals and literals in clauses[2]. In this case, a position contains two extra elements, one selecting the literal, the second the side of the (equational) literal.

Modern saturating calculi are instantiated with a *term ordering*, a partial ordering fulfilling certain properties (well-foundedness, monotonicity, compatibility with substitutions). This ordering is lifted to literals and clauses. Generating inferences can be restricted to (subterms of) maximal terms of maximal literals, and to inferences that generate at least potentially smaller consequences. Simplification allows the replacement of clauses with equivalent smaller clauses.

The most frequent generating inference used in a saturating theorem prover is *paramodulation* or its ordering-restricted variant *superposition.* In E, superposition is typically responsible for $>> 95\%$ of all generated clauses. The most important multi-clause simplifications are subsumption (which allows the removal of less general clauses if a more general clause is present) and rewriting.

In a typical saturating theorem prover, proof search is implemented with the *given-clause algorithm.* In this algorithm, the proof state is represented by two sets of clauses, the *processed clauses* P and the *unprocessed clauses* U. Initially, all clauses are unprocessed. The algorithm repeatedly picks one clause g from U, and performs all generating inferences where g is at least one premise, and all other premises are from P. The newly generated clauses are added to U, the *given clause* to P. In the DISCOUNT variant of the given-clause algorithm implemented in E, *simplification* is performed on g, and on newly generated clauses with side premises from P. Simplification of P is done with g, and clauses so simplified are removed from P and treated like newly generated clauses.

3 Fingerprint Indexing

The aim of fingerprint indexing is to organize the potentially rewritable subterms and the potential paramodulation positions in the proof state in a data structure that allows efficient retrieval of either all subterms that are matched by the left-hand side of a rewrite rule, or that can be unified with a potential partner term for paramodulation.

We will first introduce *fingerprints* of terms, and show that the compatibility of the respective fingerprints is a required condition for the existence of a unifier (or matcher) between two terms. The basic idea is that the application of a

[2] This is automatically given in most implementations, which represent literals as terms or ordered pairs, and clauses as lists.

substitution (in this case, a unifier or a matcher) never removes an existing position from a term, nor will it change an existing function symbol in a term. We can thus sample a term at a fixed number of positions, and use the collected information to find potentially unifiable or matching terms.

Consider a potential position (i.e. an arbitrary sequence of integers) p and a term t (as a running example, assume $t = g(f(x, a))$. Then the following cases are possible:

1. p is a position in t and $t|_p$ is a variable (e.g. $p = 1.1$)
2. p is a position in t and $t|_p$ is a non-variable term (e.g. $p = 1.2$ or $p = \epsilon$)
3. p is not a position in t, but there exists an instance $\sigma(t)$ with $p \in \mathrm{pos}(\sigma(t))$ (e.g. $p = 1.1.1.2$ with $\sigma = \{x \mapsto f(a, b))\}$
4. p is not a position in t or any of its instances (e.g. $p = 2.1$)

These four cases have different implications when trying to unify two terms. At the simplest, two terms which, at the same position, have different top function symbols, cannot possibly be unified. Stated positively, if we search for terms unifiable with a query term t, and $\mathrm{top}(t|_p) = f$, we only need to consider terms s where $\mathrm{top}(s|_p)$ can potentially become f.

To formalize this, consider the following definition: Let $F' = F \uplus \{\mathbf{A}, \mathbf{B}, \mathbf{N}\}$ (the set of *fingerprint feature values* for F). The *general fingerprint feature function* is a function $\mathrm{gfpf} : \mathit{Term}(F, V) \times \mathbb{N}^* \to F'$ defined by:

$$\mathrm{gfpf}(t,p) = \begin{cases} \mathbf{A} & \text{if } p \in \mathrm{pos}(t),\ t|_p \in V \\ \mathrm{top}(t|_p) & \text{if } p \in \mathrm{pos}(t),\ t|_p \notin V \\ \mathbf{B} & \text{if } p = q.r,\ q \in \mathrm{pos}(t) \text{ and } t|_q \in V \text{ for some } q \\ \mathbf{N} & \text{otherwise} \end{cases}$$

A *fingerprint feature function* is a function $\mathrm{fpf} : \mathit{Term}(F, V) \to F'$ defined by $\mathrm{fpf}(t) = \mathrm{gfpf}(t, p)$ for a fixed $p \in \mathbb{N}^*$.

Now assume two terms, s and t, and a fingerprint feature function fpf. Assume $u = \mathrm{fpf}(s)$ and $v = \mathrm{fpf}(t)$. The values u and v are *compatible for unification* if they are marked with a $\mathbf{Y}$ in the *Unification* table of Figure 1. They are *compatible for matching from s onto t*, if they are marked with a $\mathbf{Y}$ in the *Matching* table in Figure 1. It is easy to show by case distinction that compatibility of the fingerprint feature values is a necessary condition for unification or matching, respectively.[3]

Now assume $n \in \mathbb{N}$. A *fingerprint function* is a function $\mathrm{fp} : \mathit{Term}(F, V) \to (F')^n$ with the property that $\pi_n^i \circ \mathrm{fp}$ (the projection onto the ith element of the result) is a fingerprint feature function for all $i \in \{1, \ldots, n\}$. A *fingerprint* is the result of the application of a fingerprint function to a term, i.e. a vector of n elements over F'. We will in the following assume a fixed fingerprint function fp.

Two fingerprints for s and t are unification-compatible (or compatible for matching from s onto t) if they are component-wise so compatible.

[3] The only slightly unintuitive consideration encountered is the fact that application of a substitution can not only add positions to a term t, it can also restrict the set of potential positions in instances of t. As an example, consider $f(x)$. $\mathrm{gfpf}(f(x), 1.2) = \mathbf{B}$, but $\mathrm{gfpf}(f(g(a)), 1.2) = \mathbf{N}$.

Unification

	f_1	f_2	**A**	**B**	**N**
f_1	**Y**	**N**	**Y**	**Y**	**N**
f_2	**N**	**Y**	**Y**	**Y**	**N**
A	**Y**	**Y**	**Y**	**Y**	**N**
B	**Y**	**Y**	**Y**	**Y**	**Y**
N	**N**	**N**	**N**	**Y**	**Y**

Matching

	f_1	f_2	**A**	**B**	**N**
f_1	**Y**	**N**	**N**	**N**	**N**
f_2	**N**	**Y**	**N**	**N**	**N**
A	**Y**	**Y**	**Y**	**N**	**N**
B	**Y**	**Y**	**Y**	**Y**	**Y**
N	**N**	**N**	**N**	**N**	**Y**

Fig. 1. Fingerprint feature compatibility for unification and matching (down onto across). Note that f_1, f_2 stand for arbitrary function symbols, and we implicitly assume $f_1 \neq f_2$ here.

Theorem 1. *Assume an arbitrary fingerprint function* fp. *If* $\mathrm{fp}(t_1)$ *and* $\mathrm{fp}(t_2)$ *are not unification compatible, then* t_1 *and* t_2 *are not unifiable. If* $\mathrm{fp}(t_1)$ *and* $\mathrm{fp}(t_2)$ *are not compatible for matching* t_1 *onto* t_2*, then* t_1 *does not match* t_2*.*

Given a fixed fingerprint function, each term has a unique fingerprint, but many terms share the same one. A fingerprint function defines an equivalence on $\mathit{Term}(F, V)$, and we can use the fingerprints to organize any set of terms into disjoint subsets, each sharing a fingerprint. If we want to find terms in a given relation (unifiable or matchable) to a query term, we only need to consider terms from those subsets for which the query relation holds on the fingerprints.

However, we do not need to linearly compare fingerprints to find unification or matching candidates. Instead, we combine fingerprints into a *fingerprint index*. A fingerprint index is a constant-depth *trie* over fingerprints that associates the indexed term sets with the leaves (i.e. end nodes) of the trie.

As an example, consider $F = \{j|_2, f|_2, g|_1, a|_0, b|_0, e|_0\}$ and $\mathrm{fp} : \mathit{Term}(F, V) \rightarrow (F')^3$ defined by $\mathrm{fp}(t) = \langle \mathrm{gfpf}(t, \epsilon), \mathrm{gfpf}(t, 1), \mathrm{gfpf}(t, 2)\rangle$.

Figure 2 shows a fingerprint index for fp and a set of 13 terms stored at 10 leaves.[4] When we query the index for terms unifiable with $t = j(e, g(X)))$, we first compute $\mathrm{fp}(t) = \langle j, e, g\rangle$. At each node in the tree we follow all branches labeled with feature values unification-compatible with the corresponding fingerprint value of the query. At the root, only the branch labelled with j has to be considered. At the next node, branches e and **A** are compatible. Finally, three leaves (marked in darker gray), with a total of 4 terms, are unification-compatible with the query. In this case, all 4 terms found actually are unifiable with the query. The index made it possible to avoid considering any other term.

Even fairly short fingerprints are sufficient to achieve good performance of the index. Computing these small fingerprints is computationally cheap, and so is insertion and removal of fingerprints from the trie.

Since each term has a unique fingerprint, each term is stored at one and only one leaf in the trie. To find all retrieval candidates, we have to traverse the trie

[4] This is the paramodulation-from index (potentially maximal terms from potentially maximal positive literals) generated by E after 20 iterations of the main loop on the 3rd problem from [3] (a ring with $x^2 = x$ is commutative).

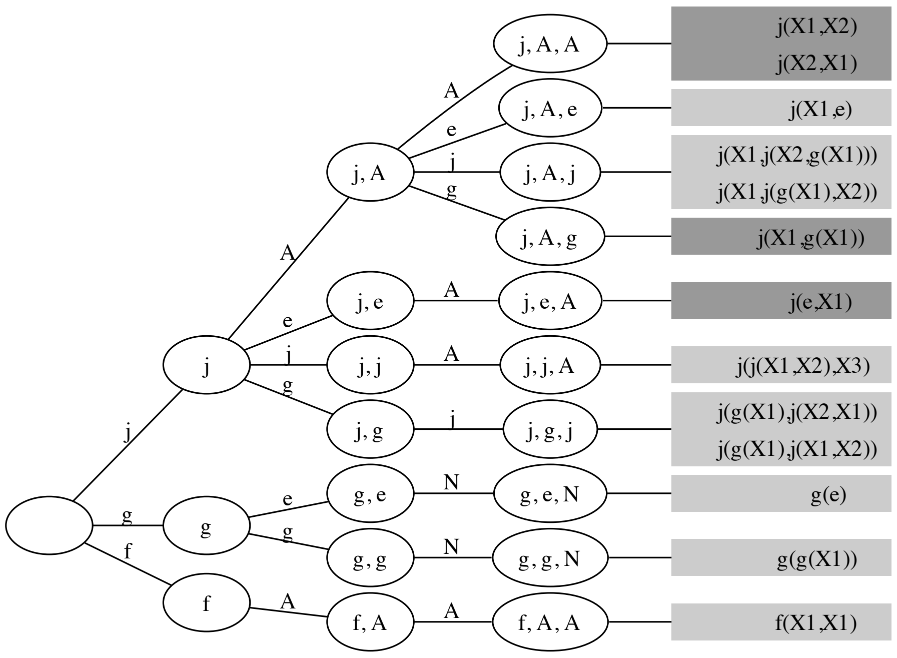

Fig. 2. Example fingerprint index

recursively, collecting candidates from all leaves. Since all terms at a leaf are compatible with all fingerprints leading to it, and since all terms are represented at most once in the index, we only need to form the union of the candidate sets at all matching leaves. This is the major advantage compared to coordinate indexing, where it is necessary to compute the intersection of all candidate sets for each coordinate. The same applies for path indexing, where e.g. Vampire goes to great lengths to optimize this bottleneck [7].

Moreover, since each term has a single fingerprint and is represented only once in the trie, there are at most as many fingerprints in an index as there are indexed terms. Thus, memory consumption of the index scales at worst linearly with the size of the set of indexed terms.

4 Implementation

As stated above, we have implemented fingerprint indexing in our theorem prover E to speed up superposition and backwards rewriting. For this purpose, we

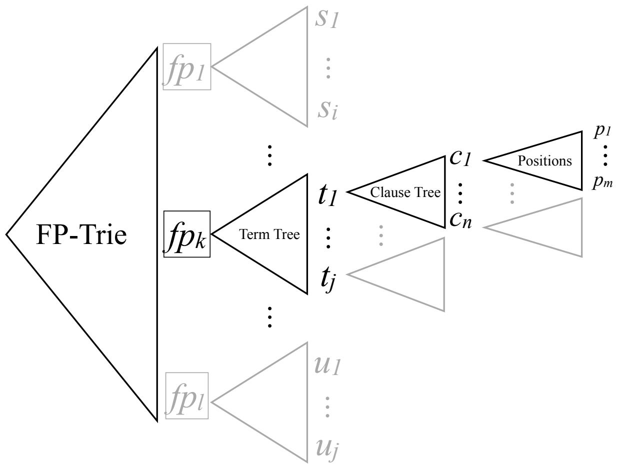

Fig. 3. Structure of the fingerprint index for paramodulation

have added three global indices to E, called the *backwards-rewriting index*, the *paramodulation-from index*, and the *paramodulation-into* index.

The *backwards-rewriting index* contains all (potentially rewritable) subterms of processed clauses. Each term is associated with the set of all processed clauses it occurs in. Given a new unit clause $l \simeq r$, we find all rewritable clauses by finding all leaves compatible with the fingerprint of l, try to match l onto each of the terms t stored at the leaf, and, in the case of success, verify if $\sigma(l) > \sigma(r)$. If and only if this is the case, all clauses associated with the term are rewritable with the new unit clause (and hence are removed from the set of processed clauses). Note that in this implementation the (potentially expensive) ordering check only has to be made once for every t, not once per occurrence t.

For the paramodulation indices, we use a somewhat more complex structure. Figure 3 visualizes the multi-level structure of the index. The FP-Trie indexes sets of terms with the same fingerprint. These sets are stored as binary trees of pointers. For each term, we store a set of clauses in which this term occurs (at a position potentially compatible with the superposition restrictions). Finally, with each of these clauses, we store the positions in which the term occurs.

The paramodulation-from index contains all maximal terms of maximal positive literals in potential side premises. To find all possible paramodulation inferences with a new clause g, we iterate over all subterms in maximal sides of maximal literals of g, and find the unifiable terms via the index. If the unification is successful with mgu σ, we can already test the ordering restrictions on $\sigma(g)$. We then iterate over all clauses associated with the term, and over all positions where the term occurs. For each, we check the ordering restrictions on the indexed clause, and, in the case of success, construct the paramodulant.

The paramodulation-into index is organized analogously, but contains all subterms of maximal terms in maximal literals of the indexed clauses. For finding paramodulants, we only iterate over the maximal terms in maximal positive literals in g. As above, we can already check ordering restrictions on g as soon as the unifier is computed, and do not have to repeat this check for every potential inference with the same term.

5 Experimental results

To measure the performance of fingerprint indexing, we performed a series of experiments. All tests use problems from the set of 15386 untyped first-order problems in the TPTP problem library [13], Version 5.2.0. The problems were not modified in any way.

A direct comparison with other indexing methods is, in principle, desirable. However, it has a number of drawbacks. E had not originally implemented any other indexing techniques for paramodulation and backward rewriting, and indeed, one of the motivations for developing lightweight techniques like feature-vector indexing and fingerprint indexing is to avoid the more complex implementation of competing techniques. Secondly, and more significantly, performance of indexing is very much also a quality-of-implementation issue. Thus, a direct comparison is not as universally useful as it would seem.

We have implemented *(non-perfect) discrimination tree indexing* [4] as a reference benchmark. In addition, we measure the performance of the index in absolute terms. For this purpose, we have instrumented the prover in a number of ways. In particular, we have added profiling code that enables us to measure and record the time spent in the relevant part of the program without the overhead and complexity of using a standard profiler. This allows us to fully automatically determine the time spent in index maintenance and unification for a large set of test problems, and and hence give an upper limit to the maximum improvements possible using any indexing technique.

5.1 Instrumenting the prover

We have added a generic profiling mechanism to E. Profiling is enabled by instrumenting the source code. The system can maintain an arbitrary number of profiling points. The system computes (at microsecond resolution) the difference between the time profiled code is entered and left. The times for each profiled segment are summed over the life time of the process. We have added timers measuring the time spent for the whole prover, the main saturation loop (excluding preprocessing), paramodulation/superposition, unification, backward rewriting, and index maintenance for paramodulation and rewriting indices.

As for all portable profiling solutions, the times measured are only statistically valid. Many functions have run times much shorter than the microsecond resolution of the UNIX system clock. However, over sufficiently many calls, the average values become increasingly more reliable. Other sources of noise include the exact scheduling of instructions on modern deeply pipelined multi-core processors with out-of-order issue, cache-contention, and memory access. Hence even the overall times measured can vary slightly from run to run. We perform tests on thousands of problems, with many thousand calls to small profiled functions for each problem. Thus, these effects largely average out. However, performance for individual, especially short-running tests, can show the effects of this noise.

5.2 Parameters

In first-order proof search, even small changes can influence the course of the proof search significantly. Since indexing affects both the order in which clauses are generated and the internal memory layout of the process, it is not always guaranteed that the system performs the same search in the different cases. To minimize the effect, we use a strategy that uses only a single simple heuristic evaluation for clauses, and impose a total order on newly generated clauses[5]. Moreover, for quantitative analysis of the run times, we only use cases where the prover has performed the same number of iterations of the main loop, and has the same number of processed and unprocessed clauses at termination time. These three indicators give a high likelihood that the proof search followed very similar lines for the indexed and non-indexed case.

This resulted in 5824 problems used for comparison. Additionally restricting the problem sets only to problems with non-trivial proof search (arbitrarily defined as having at least 1000 iterations of the main loop) did not significantly affect the results. Hence we report the result for all 5824 problems showing similar search behavior for all indexing strategies.

Tests were performed, with a number of different fingerprint functions, and with non-perfect discrimination tree indexing. *discrimination tree indexing* [4] as a benchmark. Table 1 lists the different variants.

All tests were run on the University of Miami *Pegasus cluster*. The cluster is running Linux with the 2.6.18-164.el5 SMP Kernel in 64 bit mode. Each node is equipped with 8 Intel Xeon CPUs, model L5420, running at 2.4 GHz, and 16 GB of RAM. Test runs were done with a CPU time limit of 300 seconds per job, with 8 jobs scheduled per node. The data and the version of the prover used for the test runs are archived at `http://www.eprover.eu/E-eu/FPIndexing.html`.

5.3 Results

Table 2 shows the result of the time measurements, summed over all 5824 problems. "Run time" is the total run time of the process, including preprocessing

[5] The exact options given to the prover were `--delete-bad-limit=512000000 --split-clauses=4 --split-reuse-defs --forward-context-sr --destructive-er-aggressive --presat-simplify --prefer-initial-clauses -tKBO6 -Ginvfreqconjmax -winvfreqrank -c1 -F1 -WSelectMaxLComplexAvoidPosPred -H'(1*Clauseweight(ConstPrio,1,1,1))' --detsort-new --fp-index=XXX`. We used a modified version of E 1.4.

Name	**Positions sampled**	**Remark**
NoIdx	N/A	No indexing
FP0	N/A	Fingerprint of lengths 0
FP0FP	(ϵ)	Pseudo-fingerprint emulating optimizations in the unindexed version
FP1	ϵ	Effectively top-symbol hashing
FP2	ϵ, 1	
FP3D	ϵ, 1, 1.1	
FP3W	ϵ, 1, 2	
FP4D	ϵ, 1, 1.1, 1.1.1	
FP4M	ϵ, 1, 2, 1.1	
FP4W	ϵ, 1, 2, 3	
FP5M	$\epsilon, 1, 2, 3, 1.1$	
FP6M	$\epsilon, 1, 2, 3, 1.1, 1.2$	
FP7	ϵ, 1, 2, 1.1, 1.2, 2.1, 2.2	
FP7M	ϵ, 1, 2, 3, 1.1, 4, 1.2	
FP8X2	ϵ, 1, 2, 3, 4, 1.1, 1.2, 1.3, 2.1, 2.2, 2.3, 3.1, 3.2, 3.3, 1.1.1, 2.1.1	Samples 16 positions.
NPDT	N/A (all)	Non-perfect discrimination trees

Table 1. Fingerprint functions used in the evaluation

Index	**Run time**	**Sat time**	**PM time**	**PMI time**	**MGU time**	**BR time**	**BRI time**
NoIdx	16062.392	14078.300	8980.320	0.000	2545.080	2280.250	0.000
FP0	16644.127	14835.130	9904.120	26.380	4360.330	1846.280	41.440
FP0FP	9581.606	8211.010	3633.590	27.950	1322.530	1071.210	42.030
FP1	7006.758	6145.870	1816.100	25.710	450.760	379.570	40.150
FP2	6200.043	5556.330	1345.440	28.900	199.600	104.340	43.300
FP3D	6107.780	5463.240	1266.820	31.410	150.880	91.430	46.040
FP3W	6104.037	5477.280	1264.720	32.970	149.070	75.940	46.630
FP4D	6125.753	5478.900	1253.260	32.530	138.680	89.620	50.110
FP4M	6050.617	5423.820	1197.720	33.640	109.870	64.740	49.620
FP4W	6126.202	5496.800	1261.510	33.390	149.120	75.770	50.240
FP5M	6088.364	5455.180	1203.240	38.250	107.860	65.630	53.520
FP6M	6000.177	5385.810	1181.710	38.240	99.110	39.010	55.660
FP7M	6063.286	5445.650	1189.510	39.990	99.600	38.970	58.130
FP7	6022.196	5404.150	1179.250	41.880	95.880	38.400	57.610
FP8X2	6066.482	5429.390	1193.820	56.430	88.580	37.710	77.400
NPDT	6082.246	5434.760	1184.750	64.910	83.110	33.200	79.910

Table 2. CPU times for different parts of the proof process (in seconds)

(which, with the chosen parameters, also includes a complete interreduction of the axioms with significant backwards rewriting). "Sat time" is the time spent in the main saturation loop, excluding preprocessing. "PM time" is the total time for paramodulation/superposition, including the search for inference partners, unification, ordering tests, and result assembly. Since the unification and, in the case of success, further operations, are the same for all indices, the differences in timing represent differences for the complete term retrieval for unifiers. "PMI time" is the time spent maintaining the paramodulation indices. "MGU time" is the time spent in actual unification. "BR time" and "BRI time" are the times used for backwards-rewriting and backward-rewrite index maintenance.

If we consider the total time, we can see that all proper indexing functions deliver a significant performance gain. Comparing the unindexed version with the FP6M index, total run time decreases by more than 60%. If we compare the first (unindexed) and the third row (indexed using a pseudo-fingerprint equivalent to some optimizations used in the non-indexed implementation), we can see that even here the unification time is cut in half.Moreover, the indexed variant gains a further 5000 seconds in paramodulation. The first effect shows the effect of performing unification only once per term, not once per term occurrence. The second one shows the even more significant effect of performing the ordering checks of superposition on the query clause only once, when the instance is generated, and not repeatedly for every inference pair.

The time spent for unification itself has been reduced by a factor of about 25 using the FP6M index. With this index, the total time for unification-related code (i.e. index maintenance and unification) amounts to less than 2.5% of the total run time.

Comparing the times for FP6M with the times for discrimination tree indexing, we see that overall fingerprint indexing outperforms discrimination tree indexing, if not by a large margin. Time for actual unification is slightly lower for discrimination tree indexing, however, because of the indexes greater size, index maintenance and index traversal for retrieval are more expensive than for fingerprint indexing.

Table 3 shows the success of fingerprint indexing in delivering good unification and matching candidates. Note that in the non-indexed case, every distinct occurrence of a term will potentially be tried for unification and matching. For all indices, terms are represented at most once in the index. Thus, the number of successful unifications is lower, although the number of successful inferences is the same. The fact that all indexed strategies have exactly the same number of successful unifications and matches is another strong indicator that the prover executed the same proof search for each different strategy.

In the unindexed case, only about 2% of all unification attempts are successful. Even for short fingerprints, the ratio increases significantly, and for the FP6M strategy, more 60% of all candidate terms do unify. This ratio increases further as fingerprints become longer. For non-perfect discrimination trees, it reaches about 75%.

We see an even stronger improvement for backwards rewriting. Here, the time for the operation itself drops more than 58-fold. Time for index maintenance is of the same order of magnitude as time for matching. This is not surprising, since the backwards-rewrite index needs to store all subterms, while the paramodulation indices only have to store potential inference terms. Taking index maintenance into account, the total time for backward rewriting improved by a factor of about 25. The time spent for backwards-rewriting and index main-

Index	**Unification attempts**	**Unification successes**	**Unification succ. rate**	**BR match attempts**	**BR match successes**	**BR Match succ. rate**
NoIdx	9 540 075 320	192 664 262	0.020	9 658 509 219	156 698	0.000
FP0	17 211 802 533	90 278 056	0.005	9 586 440 451	144 337	0.000
FP0FP	4 738 930 478	90 278 056	0.019	5 834 612 493	144 337	0.000
FP1	1 163 944 204	90 278 056	0.078	2 285 521 238	144 337	0.000
FP2	383 514 084	90 278 056	0.235	367 303 091	144 337	0.000
FP3D	280 445 270	90 278 056	0.322	271 434 503	144 337	0.001
FP3W	243 476 462	90 278 056	0.371	236 615 956	144 337	0.001
FP4D	265 042 869	90 278 056	0.341	252 155 842	144 337	0.001
FP4M	165 525 195	90 278 056	0.545	165 095 233	144 337	0.001
FP4W	239 979 381	90 278 056	0.376	235 742 889	144 337	0.001
FP5M	162 238 728	90 278 056	0.556	164 298 660	144 337	0.001
FP6M	145 567 515	90 278 056	0.620	53 841 789	144 337	0.003
FP7M	145 509 054	90 278 056	0.620	53 745 992	144 337	0.003
FP7	140 261 300	90 278 056	0.644	46 773 678	144 337	0.003
FP8X2	126 341 321	90 278 056	0.715	33 611 032	144 337	0.004
NPDT	118 072 345	90 278 056	0.765	561 473	144 337	0.257

Table 3. Retrieval attempts and successes

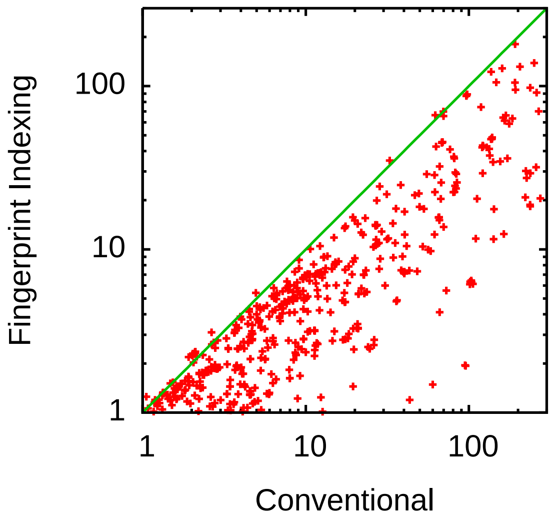

Fig. 4. Scatter plot of unindexed and FP6M total run times (in seconds)

tenance for FP6M is only 1.6% of the total time, down from about 15% for the non-indexed version.

We can observe that discrimination tree indexing does further improve the times for actual term retrieval, but performs worse overall due to the higher cost of index maintenance.

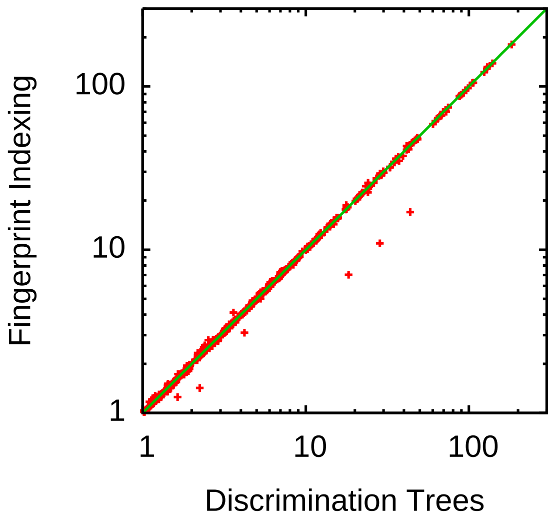

Fig. 5. Scatter plot of discrimination tree and FP6M total run times (in seconds)

Figure 4 shows a scatter plot of run times for the sequential search implementation and FP6M fingerprint indexing. Only problems with a run time of at least one second are included. Please note the double logarithmic scale. For the vast majority of problems, the indexed version is significantly, and often dramatically, faster, while there are no problems for which the conventional version is more than marginally faster.

Figure 5 compares FP6M and discrimination tree indexing. We can see that for most problems, performance is nearly the same, but that for a few problems, discrimination tree indexing is distinctly slower. There are no cases where fingerprint indexing is significantly slower.

6 Conclusion

In this paper, we have introduced *fingerprint indexing*, a lightweight indexing technique that is easy to implement, has a small memory footprint, and shows excellent performance in practice. It is particularly effective for typical application problems, with large signatures and large numbers of axioms. Fingerprint indexing has made E much more competitive on this kind of proof problems.

In the future, we will further investigate the influence of different fingerprint functions, and evaluate if further gains can be made by automatically generating a good fingerprint function based on the signature.

While the experiments show that fingerprint indexing has the potential to reduce the CPU cost of the indexed operations to nearly insignificant levels, we are still interested in a direct comparison with additional indexing techniques in a synthetic setting like the one discussed in [5].

Acknowledgements: I thank the University of Miami's Center for Computational Science HPC team for making their cluster available for the experimental evaluation.

References

1. Graf, P.: Term Indexing, LNAI, vol. 1053. Springer (1995)
2. Graf, P., Fehrer, D.: Term Indexing. In: Bibel, W., Schmitt, P. (eds.) Automated Deduction — A Basis for Applications, Applied Logic Series, vol. 9 (2), chap. 5, pp. 125–147. Kluwer Academic Publishers (1998)
3. Lusk, E., Overbeek, R.: A Short Problem Set for Testing Systems that Include Equality Reasoning. Tech. rep., Argonne National Laboratory, Illinois (1982)
4. McCune, W.: Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. Journal of Automated Reasoning 9(2), 147–167 (1992)
5. Nieuwenhuis, R., Hillenbrand, T., Riazanov, A., Voronkov, A.: On the Evaluation of Indexing Techniques for Theorem Proving. In: Goré, R., Leitsch, A., Nipkow, T. (eds.) Proc. of the 1st IJCAR, Siena. LNAI, vol. 2083, pp. 257–271. Springer (2001)
6. Riazanov, A., Voronkov, A.: The Design and Implementation of VAMPIRE. Journal of AI Communications 15(2/3), 91–110 (2002)
7. Riazanov, A., Voronkov, A.: Efficient Instance Retrieval With Standard and Relational Path Indexing. In: Bader, F. (ed.) Proc. of the 19th CADE, Miami. LNAI, vol. 2741, pp. 380–396. Springer (2003)
8. Schulz, S.: E – A Brainiac Theorem Prover. Journal of AI Communications 15(2/3), 111–126 (2002)
9. Schulz, S.: Simple and Efficient Clause Subsumption with Feature Vector Indexing. In: Sutcliffe, G., Schulz, S., Tammet, T. (eds.) Proc. of the IJCAR-2004 Workshop on Empirically Successful First-Order Theorem Proving, Cork, Ireland (2004)
10. Schulz, S.: System Description: E 0.81. In: Basin, D., Rusinowitch, M. (eds.) Proc. of the 2nd IJCAR, Cork, Ireland. LNAI, vol. 3097, pp. 223–228. Springer (2004)
11. Sekar, R., Ramakrishnan, I., Voronkov, A.: Term Indexing. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. II, chap. 26, pp. 1853–1961. Elsevier Science and MIT Press (2001)
12. Stickel, M.E.: The Path-Indexing Method for Indexing Terms. Technical Note 473, Artificial Intelligence Center, SRI International, Menlo Park, California, USA (October 1989)
13. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. Journal of Automated Reasoning 43(4), 337–362 (2009)
14. Weidenbach, C., Schmidt, R., Hillenbrand, T., Topić, D., Rusev, R.: SPASS Version 3.0. In: Pfenning, F. (ed.) Proc. of the 21st CADE, Bremen. LNAI, vol. 4603, pp. 514–520. Springer (2007)