Computing All Implied Equalities via SMT-based Partition Refinement

Josh Berdine and Nikolaj Bjørner

Microsoft Research
{jjb, nbjorner}@microsoft.com

Abstract. Consequence finding is used in many applications of deduction. This paper develops and evaluates a suite of optimized SMT-based algorithms for computing equality consequences over arbitrary formulas and theories supported by SMT solvers. It is inspired by an application in the SLAYER analyzer, where our new algorithms are commonly 10–100x faster than simpler algorithms. The main idea is to incrementally refine an initially coarse partition using models extracted from a solver. Our approach requires only $O(N)$ solver calls for N terms, but in the worst case creates $O(N^2)$ fresh subformulas. Simpler algorithms, in contrast, require $O(N^2)$ solver calls. We also describe an asymptotically superior algorithm that requires $O(N)$ solver calls and only $O(N \log N)$ fresh subformulas. We evaluate algorithms which reduce the number of fresh formulas required either by using specialized data structures or by relying on subformula sharing.

Keywords: Implied Equalities · Consequence Finding · Satisfiability Modulo Theories · Decision Procedures · Congruence Closure · Software Verification

1 Introduction

We define and evaluate optimized algorithms for computing all equalities between terms of a fixed set that are implied by a fixed constraint. As a example, consider the formula

$$\Phi : (a \simeq b \wedge b[i] \simeq c) \vee (a[i]{-}4 \simeq d \wedge f(d) \simeq d{+}3 \wedge f(d){+}1 \simeq c)$$

where _[_] denotes array selection. Φ implies the equality $a[i] \simeq c$, but not $a \simeq b$, $a[i] \simeq b[i]$, nor $d \simeq c$. On the other hand, the formula

$$\Phi' : \Phi \vee (b[i] \simeq c \wedge c \simeq d)$$

does not imply any equality between $a, b, c, d, a[i]$ and $b[i]$. We describe and evaluate algorithms that require a number of SMT-solver calls that is at worst linear in the number of terms N, although they may potentially create quadratically-many fresh literals. Naïve algorithms for computing all equalities implied by a formula require $O(N^2)$ SMT-solver calls. The simplest, called *Basic Partition Merging* (BPM), is 10–1000x slower than a model-based variant, *Model-based Partition Merging* (MPM). Starting with *Basic* (BPR) and *Incremental* (IPR) *Partition Refinement* algorithms, we examine several variants that are all significantly (1–100x) faster than MPM. In a quest of asymptotically better solutions, we also outline an algorithm that requires only $O(N \log N)$ fresh subformulas.

Application and Evaluation of Equality Inference The experimental evaluation of the algorithms uses problem instances that the SLAYER [1] program analyzer encounters while attempting to verify memory safety properties of C programs. SLAYER relies on learning implied equalities, and previously used simpler algorithms that would sometimes exhibit catastrophic performance. This prompted the development of more sophisticated algorithms. We evaluated both the folklore and the new algorithms in the context of the SLAYER tool and found that the simpler algorithms are impractical because they either require a quadratic number of solver calls, or are non-incremental: they effectively reset the solver state between several calls. The practical evaluation demonstrates the advantages of the new algorithms.

Consequence finding is a central component of abstract interpreters. The set of reachable states can be approximated by the least fixed point of a predicate transformer, and analyses based on abstract interpretation commonly develop special classes and representations of logical formulas where approximations of the least fixed point can be computed. For example, analyses using the octagon abstract domain [9] can be thought of as computing consequences as a conjunction of constraints using unit coefficients and two variables per inequality. The TVLA system [8, 14] is distinguished as it produces shape formulas as a result of bottom-up evaluation of Horn clauses.

The SLAYER tool synthesizes separation logic formulas in an approximation of the least fixed-point semantics of programs. This approach serves as a foundation for a verification tool for analyzing heap properties of C programs. In order to compute precise abstractions, SLAYER relies on learning all implied equalities from a formula. An example symbolic state is the formula $\exists x, y.\ \Phi * \Psi$, where Φ is defined above and

$$\Psi : \mathsf{list}(p, x) * x{\mapsto}c * \mathsf{list}(q, y) * y{\mapsto}a[i]\ .$$

SLAYER weakens the formula to $\mathsf{list}(p, c) * \mathsf{list}(q, c)$. The first step of this abstraction is to replace $\mathsf{list}(p, x) * x{\mapsto}c$ with $\mathsf{list}(p, c)$, forgetting that x is on the list from p to c. This rewrite would not be performed if x could begin a shared tail between two lists. Checking this requires learning that neither $x \simeq q$, $x \simeq y$ nor $x \simeq a[i]$ are implied, any of which would make x begin a tail shared between p and q. Finally, $\mathsf{list}(q, y) * y{\mapsto}c$ is rewritten to $\mathsf{list}(q, c)$. This inference requires learning $a[i] \simeq c$, as previously discussed.

Note that the core of checking the shared-tail condition is to compare the sets of predecessors in the transitive closure of the equivalence closure of the union of the $\mapsto$ and list relations. The use of equivalence closure necessitates an eager computation of the equality consequences of formulas.

Related Work Classical congruence closure [5] infers equalities from conjunctions of equalities with uninterpreted functions. In contrast, the problem addressed here is to infer congruences modulo arbitrary formulas (e.g., clauses instead of conjunctions), over theories supported by SMT solvers. Satisfiability Modulo Theories solvers that are based on the DPLL(T) architecture [11] use congruence closure to check satisfiability of a conjunction of assumed equalities with respect to assumed disequalities. Saturation based theorem provers for classical first-order logic use superposition inference rules to deduce new equalities from old ones. The algorithmic problems of unification and congruence closure have received significant attention and enjoyed several celebrated

results, such as linear time algorithms for unification [12, 13] and efficient congruence closure algorithms [5]. Saturation procedures rely on term indexing and unification algorithms and higher performance saturation engines strike a trade-off between indexing data structures and the unification algorithms that are used in practice [7]. Similarly, SMT solvers use congruence closure algorithms that have shown to perform well in their context of use [4, 10]. For instance, Z3's congruence closure algorithm uses a variant of union-find with eager path compression.

Equality inference of Boolean functions is a deeply studied subject in the context of circuit verification [2, 3] because when checking equality between two circuits it can be a significant advantage to know that two sub-circuits compute the same Boolean function. Some of the main techniques use binary decision diagrams, BDDs, for identifying sub-circuit equivalence. Note that equivalence of Boolean functions is a special case of the problem we consider here: we are determining equivalence of not only Boolean functions, but functions with any signature (e.g., functions over reals and integers). Additionally, for the sub-circuit equivalence problem, it suffices to find enough equalities to speed up the equivalence check, while we consider the problem of computing the complete set. Detecting and using equivalences can also have profound effects on SAT solving [6]. The use of SAT sweeping is critical for reducing the set of candidate equivalences. SAT sweeping and possible generalizations to SMT provides orthogonal value to the algorithms developed here.

Organization Section 2 first recalls a few technical preliminaries. Section 3 discusses the simpler algorithms and then presents the Basic and Incremental Partition Refinement algorithms and variants. The practical context where the algorithms are used is discussed in Section 4. Section 5 closes with a thorough evaluation.

2 Preliminaries

We assume some basic familiarity with SMT solving and, to a greater degree, congruence closure.

SMT, Models and Formulas We use standard notions of sorts, terms, formulas and interpretations. Formulas are terms of Boolean sort. Terms are built using a first-order signature where functions, such as $+$ and $\_[\_]$ can be interpreted. We assume that interpretations, denoted $\mathcal{M}$ can be used to evaluate terms to values. For example, if $\mathcal{M} \models x+2 \simeq 1$, then $\mathcal{M}(x) = -1$.

Union-Find Our algorithms maintain partitions and use the well-known union and find routines [15]. Given a domain E of nodes and a domain $\mathcal{P}$ of partitions of E, a partition P of $\{1, \ldots, N\}$ is a set of disjoint subsets, called classes, that cover the set. We assume the following routines:

- $\mathit{find} : \mathcal{P} \times E \to E$, such that $\mathit{find}(P, i)$ returns a unique representative from the equivalence class of i. Thus, $\mathit{find}(P, \mathit{find}(P, i)) = \mathit{find}(P, i)$.
- $\mathit{union} : \mathcal{P} \times E \times E \to \mathcal{P}$, such that the equivalence classes of i and j are merged in the result of $\mathit{union}(P, i, j)$. Thus $\mathit{find}(\mathit{union}(P, i, j), i) = \mathit{find}(\mathit{union}(P, i, j), j)$.

We furthermore use a nonstandard routine to split equivalence classes by removing an element from a class and creating a new singleton class:

- $\mathit{remove} : \mathcal{P} \times E \to \mathcal{P}$, such that $\mathit{find}(\mathit{remove}(P, i), j) = i$ if and only if $i = j$.

We only need to call $\mathit{remove}(P, i)$ in the case when i is not an equivalence class representative. So if we implement *union* using eager path compression, then *remove* is realized by detaching the removed node from a doubly-linked list.

3 Algorithms for Implied Equalities

The problem of computing all implied equalities can be stated as: given a formula Φ and a set of terms $t_1, \ldots, t_N$, find a partition P of $\{1, \ldots, N\}$, such that for every $p, q \in P$ and $s \in p, t \in q$: $(\Phi \to s \simeq t)$ is valid if and only if $p = q$. To set the stage for our partition refinement algorithms, we begin by discussing some simpler algorithms.

3.1 Basic Partition Merging (BPM)

The most straightforward approach for finding all implied equalities is to check whether an equality is implied for each pair of terms. In more detail, create a partition P of the indices $\{1, \ldots, N\}$ by checking whether $\Phi \to t_i \simeq t_j$ for each pair $1 \leq i < j \leq N$. One can save a redundant check if j is already merged with another index $k < j$. The union-find data structure can be used to maintain the partition as it is built. Asymptotically, this straightforward algorithm requires $O(N^2)$ solver calls in the worst case.

Example 1 Consider again the formula $\Phi : (a \simeq b \wedge b[i] \simeq c) \vee (a[i]{-}4 \simeq d \wedge f(d) \simeq d{+}3 \wedge f(d){+}1 \simeq c)$. We wish to partition the terms $\{a, b, c, d, a[i], b[i]\}$ in the context of Φ. Only the formula $\Phi \to a[i] \simeq c$ is valid, so the resulting partition is $\{\{a\}, \{b\}, \{c, a[i]\}, \{d\}, \{b[i]\}\}$. Other validity checks fail. Once $a[i]$ and c are found to be equal, it is redundant to check both $\Phi \to a[i] \simeq d$ and $\Phi \to c \simeq d$. ⌟

3.2 Model-based Partition Merging (MPM)

The previous algorithm uses very little information between solver calls. More useful information is available when the SMT solver produces *models*. Whenever checking that $\Phi \to t_i \simeq t_j$ is valid, we do in fact check dually whether $\Phi \wedge t_i \not\simeq t_j$ is unsatisfiable. If it is satisfiable, then a model can be extracted that satisfies Φ and the disequality $t_i \not\simeq t_j$. The model may also distinguish other terms that are not tested for equality. So the set of *potential* equalities that have to be tested can be reduced by inspecting the model after each satisfiability check, exploiting the property that if two terms evaluate to distinct values in a model, their equality cannot be implied. While this algorithm still requires $O(N^2)$ solver calls, all but one of them will be satisfiable, in contrast to BPM where each merge is the result of an unsatisfiable query. This is relevant since solvers often can solve satisfiable queries faster than unsatisfiable ones.

Example 2 Continuing with Φ, suppose that it has a model where $i = 0$, $a[i] = b[i] = c = 1$, $d = a[1] = 2$, $b[1] = 3$. We then know that it only makes sense to check equalities among $\{a[i], b[i], c\}$ instead of the full set including $\{d, a, b\}$. ⌟

3.3 Basic Partition Refinement (BPR)

We saw that models play a role dual to validity checks: they indicate what equalities are *not* implied. We can take this idea to its fullest extent and develop an algorithm that splits partitions based on models instead of merging partitions based on validity checks. If we ensure that every satisfiability check splits at least one class, then this approach requires at worst only a linear number of solver calls. To start with, we can check satisfiability of $\Phi \wedge \bigvee_{i>1} t_1 \not\simeq t_i$. Any model of the formula must satisfy at least one disequality. The model also produces a more refined partition: only terms that are equal in the current model have to be compared for disequality. So suppose P is the current partition, then we define the formula

$$\mathit{SomeDiff} : \bigvee_{p \in P,\ |p|>1,\ i \in p} \ \bigvee_{j \in p,\ i \neq j} t_i \not\simeq t_j$$

such that $\Phi \wedge \mathit{SomeDiff}$ is satisfiable if and only if some class can be split.

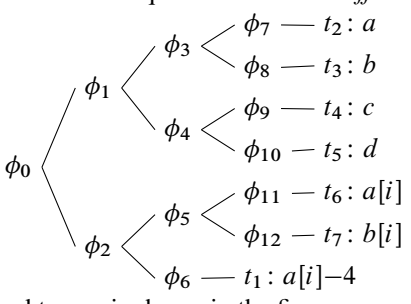
Example 3 Continuing the previous example, we create the predicate $\mathit{SomeDiff} : c \not\simeq a[i] \vee c \not\simeq b[i] \vee a \not\simeq b$ corresponding to the partition $\{\{c, a[i], b[i]\}, \{d\}, \{a, b\}\}$. The formula $\Phi \wedge \mathit{SomeDiff}$ is satisfiable where $c \not\simeq b[i]$ is true, $c \not\simeq a[i]$ is false (so $c \simeq a[i]$), and $a \not\simeq b$ is true. The next formula $\Phi \wedge c \not\simeq a[i]$ is unsatisfiable, so we learn that $c \simeq a[i]$ is implied. ⌟

This method only requires a linear number of solver calls. Yet, it is either non-incremental (the *SomeDiff* constraint is retracted between each solver call), or it requires quadratic space: it has to create fresh disequality literals between subsequent calls and has to create new clauses at every call. That is, it is unclear whether the complexity has simply been shifted around, or in other words, if $O(N)$ solver calls which each receive $O(N)$ new constraints is an improvement over $O(N^2)$ solver calls which each receive $O(1)$ new constraints. We will now develop algorithms that address time and space deficiencies of BPR.

3.4 Incremental Partition Refinement (IPR)

The IPR algorithm refines BPR by allowing reuse of literals and clauses between iterations. The algorithm creates a binary heap of propositional variables. The heap has a leaf for each term, where the proposition is constrained to hold if and only if the term is not equal to its representative. The proposition of each internal node holds if and only if one of its children does. The root of this heap is therefore equivalent to *SomeDiff*.

Example 4 Consider computing the equality partition of terms $\{a[i]{-}4, a, b, c, d, a[i], b[i]\}$ implied, again, by the formula $\Phi : (a \simeq b \wedge b[i] \simeq c) \vee (a[i]{-}4 \simeq d \wedge f(d) \simeq d{+}3 \wedge f(d){+}1 \simeq c)$. The algorithm starts with the initial partition $P = \{\{1, \ldots, 7\}\}$ and initializes a proposition heap ϕ of size 12, adding the leaf and internal node constraints. The tree structure induced, as well as the associations between propositions ϕ_i and nodes, and between leaves and terms, is shown in the figure. ⌟



The heap is used in several ways. The first use we describe is to use the truth values of the propositions in a model $\mathcal{M}$ to determine what partitions to refine.

Example 5 Continued: Φ conjoined with the proposition heap constraints is satisfiable, and suppose the extracted model $\mathcal{M}$ satisfies $a[i]{-}4 \simeq d$, $c \simeq a[i] \simeq b[i]$, $a \simeq b$, $c \not\simeq d$, $a \not\simeq d$ and $a \not\simeq c$. Therefore (assuming for the sake of presentation, that the *find* routine returns the element t_i of a class with minimal i) $\mathcal{M}$ will assign ϕ_6 and ϕ_{10} to *false* and the other propositions to *true*. This is because the term associated with ϕ_6, $t_1 = a[i]{-}4$ was the representative of the single class, and ϕ_{10} is associated to $t_5 = d$, which is still equal to its representative $a[i]{-}4$ in $\mathcal{M}$. All other terms are no longer equal to their existing representative. ⌟

The heap is only updated along paths where a model $\mathcal{M}$ established that candidate equalities were found to be not implied.

Example 6 Continued: The constraints for the new partition are constructed by updating the proposition heap using a depth-first traversal starting from the root 0, excluding sub-trees whose propositions do not hold. Since $\mathcal{M}(\phi_i) = \mathit{true}$ for all i except $6, 10$, traversal proceeds from the root 0 to the leaf 7. To update the heap at 7, first the partition is refined to $\mathit{remove}(P, 2)$ by removing the associated term, $t_2 = a$, from its current class. Then ϕ_7 is overwritten with a fresh proposition, and the updated partition is used to conjoin $\phi_7 \leftrightarrow t_2 \not\simeq t_{\mathit{find}(P,2)}$ for the fresh ϕ_7 to Φ, reestablishing the leaf constraint. Next, 8, associated with $t_3 = b$, is visited. This proceeds similarly to the update for 7, except that now, since $\mathcal{M}$ satisfies $a \simeq b$, the partition is updated to $\mathit{union}(\mathit{remove}(P, 3), 3, 2)$, removing b from its current class and merging it into the class of a. The updated partition is again used to extend Φ to reestablish the leaf constraint for a fresh ϕ_8. Next, to update the internal node 3, ϕ_3 is overwritten with a fresh proposition and Φ is extended to reestablish the internal constraint $\phi_3 \leftrightarrow (\phi_7 \vee \phi_8)$ for the new propositions. Updating the heap proceeds similarly until reaching 10. Since $\mathcal{M}(\phi_{10}) = \mathit{false}$, ϕ_{10} and its existing constraints are reused. Updating then proceeds similarly, refining the partition to $P = \{\{1, 5\}, \{2, 3\}, \{4, 6, 7\}\}$. ⌟

Note how the leaf constraints, where each term is disequal to its representative, mirror an eagerly path-compressed union-find representation of the current partition. This design is significant since it ensures that leaf propositions for representatives are always inconsistent (representatives are their own representatives), and hence an equivalence class representative will never be chosen for removal from its current class. Without this guarantee, the simple implementation of the nonstandard *remove* routine described in Section 2 would not suffice.

Algorithm 1 distills the examples as the IPR algorithm. The binary heap of propositional variables is represented as an array, which is initialized with fresh propositional variables on line 3. Line 4 then adds the heap constraints described above to the input formula. Following the approach illustrated by the preceding examples, the *update* procedure uses the values of the ϕ_i to traverse the part of the binary heap containing satisfied disequalities, constructing new disequalities for the leaves where the partition changes, and rebuilding the binary heap of propositions reusing as many subformulas as possible, so that ϕ_0 is again equivalent to the disjunction of the disequalities between each term and its current representative. Incrementally updating the partition in

Algorithm 1: Incremental Partition Refinement (IPR)

Input: formula Φ and set of terms $\{t_1, \ldots, t_N\}$
Output: equality partition P of the set $\{1, \ldots, N\}$

1 $P \leftarrow \{\{1, \ldots, N\}\}$
2 **foreach** $i = 0 \ldots 2N-2$ **do**
3 $\quad \phi_i \leftarrow$ fresh propositional variable
4 $\Phi \leftarrow \Phi \wedge \phi_0 \wedge \bigwedge_{i=0}^{N-2} (\phi_i \leftrightarrow (\phi_{2i+1} \vee \phi_{2i+2})) \wedge \bigwedge_{i=1}^{N} (\phi_{i+N-2} \leftrightarrow t_i \not\simeq t_{find(P,i)})$
5 **while** Φ *is satisfiable* **do**
6 $\quad \mathcal{M} \leftarrow$ interpretation satisfying Φ
7 $\quad Q \leftarrow \emptyset$
8 $\quad$ **Procedure** *update*(i) **is**
9 $\quad\quad$ **if** $\mathcal{M}(\phi_i) = true$ **then**
10 $\quad\quad\quad \phi_i \leftarrow$ fresh propositional variable
11 $\quad\quad\quad$ **if** $i < N-1$ **then** // i is internal
12 $\quad\quad\quad\quad$ *update*$(2i+1)$
13 $\quad\quad\quad\quad$ *update*$(2i+2)$
14 $\quad\quad\quad\quad \Phi \leftarrow \Phi \wedge (\phi_i \leftrightarrow (\phi_{2i+1} \vee \phi_{2i+2}))$
15 $\quad\quad\quad$ **else** // i is a leaf
16 $\quad\quad\quad\quad$ **let** $j = i - (N-2)$ // leaf i is associated with term t_j
17 $\quad\quad\quad\quad$ **let** $k =$ *find*(P, j)
18 $\quad\quad\quad\quad$ **assert** $k \neq j$
19 $\quad\quad\quad\quad P \leftarrow$ *remove*(P, j)
20 $\quad\quad\quad\quad$ **if** $\langle k, \mathcal{M}(t_j)\rangle \notin$ **dom** Q **then**
21 $\quad\quad\quad\quad\quad Q[\langle k, \mathcal{M}(t_j)\rangle] \leftarrow j$
22 $\quad\quad\quad\quad$ **else**
23 $\quad\quad\quad\quad\quad$ **let** $h = Q[\langle k, \mathcal{M}(t_j)\rangle]$
24 $\quad\quad\quad\quad\quad P \leftarrow$ *union*(P, j, h)
25 $\quad\quad\quad\quad \Phi \leftarrow \Phi \wedge (\phi_i \leftrightarrow t_j \not\simeq t_{find(P,j)})$
26 $\quad$ *update*(0)
27 $\quad \Phi \leftarrow \Phi \wedge \phi_0$
28 **return** P

lines 18–24 uses a temporary map Q from pairs of (representatives of) classes of the previous partition and values to classes of the updated partition. This map is used to merge terms in the updated partition that are both in the same class of the previous partition and given the same value by $\mathcal{M}$. This is necessary to avoid the refined P breaking more equalities than $\mathcal{M}$ refuted. Lines 10–25 are executed for each j that should change class, that is, such that $\mathcal{M}$ satisfies $t_j \not\simeq t_{find(P,j)}$. First j is removed from its existing class. (Note that since $\mathcal{M}$ satisfies $t_j \not\simeq t_{find(P,j)}$, $j \neq find(P, j)$, so the *remove* operation is not problematic.) Then if there is not already another class for terms of j's previous class, k, and current value, $\mathcal{M}(t_j)$, then record j as such a class. Otherwise, merge j into the existing class h. Therefore, the effect of lines 26–27 is to

strictly refine P while preserving all equalities that are true in $\mathcal{M}$, and to extend Φ to admit only models which violate the new P.

Example 7 Continued: The map Q is initially empty, so when node 7, associated with $t_2 = a$, of the heap is updated, Q is updated with a mapping from $\langle 1, \mathcal{M}(a)\rangle$ to 2 to record that 2 is the new representative for terms currently in the class $\{\{1, \ldots, 7\}\}$ that are also equal to a in $\mathcal{M}$. Then, when 8, associated with $t_3 = b$, is updated, since $\mathcal{M}$ satisfies $a \simeq b$, Q contains a mapping for $\langle 1, \mathcal{M}(b)\rangle$. So P is updated to merge the classes of 3 and 2.

Continuing with the second iteration, $P = \{\{1, 5\}, \{2, 3\}, \{4, 6, 7\}\}$, and Φ is still satisfiable. Suppose the new model $\mathcal{M}$ still satisfies $a[i] \simeq c$, and now satisfies $a \not\simeq b$, $a[i]{-}4 \not\simeq d$ and $c \not\simeq b[i]$, as well as $d \simeq b[i]$. Therefore $\mathcal{M}$ will satisfy $\phi_8, \phi_{10}, \phi_{12}$, as well as their ancestors, but no others. The updates for 8 and 10 will proceed similarly to the first iteration described above. The update for 12 is similar, but illustrates the necessity of keying the map Q on pairs of the *current class and* value. Note that $\mathcal{M}$ equates $t_5 = d$ and $t_7 = b[i]$, but t_5 and t_7 are in distinct classes of P. Therefore when updating 12, the lookup at line 20 does not find an existing entry, and so does not merge the classes of 7 and 5. Keying Q on only the term values would result in merging classes that have already been split. This would violate the property that the partition is monotonically refined, which is crucial to making only a linear number of solver calls.

Finally at the end of the second iteration, $P = \{\{1\}, \{2\}, \{3\}, \{4, 6\}, \{5\}, \{7\}\}$, and Φ is unsatisfiable. ⌟

Compared to the BPR algorithm, Algorithm 1 constructs fewer, but still $O(N^2)$, new disequalities. The problem is that terms may change representative many times. Consider a case where no equalities are implied, and an execution where at iteration $i \in 1 \ldots N$ the interpretation satisfies all disequalities $t_j \not\simeq t_{\mathit{find}(P,j)}$ for $i < j \leq N$. Therefore at iteration i, $N - i$ new disequalities will be created, and overall $\sum_{i=1}^{N} N - i = \frac{1}{2}N(N-1)$ disequalities are created. We here use problem instances encountered in practice to justify that this worst-case scenario is unlikely.

Assumption-based IPR (ABIPR) We also experimented with a slight variation of IPR that uses assumptions to control the contents of leaf nodes. The variant uses fresh propositional variables a_i and assertions of the form $a_i \rightarrow (\phi_i \leftrightarrow t_i \not\simeq t_{\mathit{find}(P,i)})$. With these constraints conjoined to Φ, the satisfiability check is replaced with a satisfiability check subject to also assuming the a_i for each leaf. This avoids accumulating assertions and is potentially more incremental. Indeed our experimental evaluation did show improvements in performance for this alternative over IPR, yet the improvements were not major for our evaluation suite.

Incrementality via Term Sharing (HIPR) Some solvers ensure maximal sharing of terms, that is, if a term that occurs in an existing constraint is constructed, the existing term is reused for the newly constructed one. For such solvers, asserting a constraint that is, e.g., a disjunction of N disequations where most already occur in existing constraints is not significantly more expensive than asserting only the new disequations. Native support for n-ary disjunction is also beneficial in this situation.

We experimented with a hybrid incremental partition refinement (HIPR) algorithm that is a hybrid between BPR and IPR. Like BPR, it asserts a disjunction of N disequations at each refinement iteration. The particular disequations are those at the leaves of IPR's proposition heap, thereby ensuring that many disequations will be shared with those from the previous constraints. In this way, much of the benefit of IPR's incrementality may be realized without the overhead of manipulating the proposition heap. Indeed, our experimental evaluation indicates that, for our evaluation suite, the overhead of manipulating the propositions sometimes significantly outweighs the overhead of repeatedly reconstructing existing disequations.

3.5 Space-Optimized Partition Refinement

We can do even better asymptotically. In the following we outline, omitting lower-level details, a partition refinement based algorithm that takes $O(N)$ iterations and has an overhead of $O(N \log N)$ fresh sub-terms.

The idea is as follows: Similar to Algorithm 1 we will maintain a binary tree rooted in ϕ_0 that covers the current disjunction of disequalities. In contrast to that algorithm, however, we represent each class of a partition as a disjunction *chain* of the form

$$t_1 \not\approx t_2 \vee t_2 \not\approx t_3 \vee \cdots \vee t_{N-1} \not\approx t_N \tag{1}$$

instead of a disjunction *star* of the form

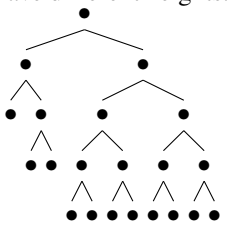
$$t_1 \not\approx t_j \vee t_2 \not\approx t_j \vee \cdots \vee t_N \not\approx t_j$$

where $1 \leq j \leq N$ is the equivalence class representative.

We maintain a two level binary tree where internal nodes are labeled by literals ϕ_i and constraints of the form $\phi_i \leftrightarrow (\phi_{2i+1} \vee \phi_{2i+2})$.

The lower level summarizes the disjunction of asserted disequalities within a current class. The upper level summarizes the disjunction of disequalities for classes of size at least 2. Let us consider the case where one of the classes is refined. So suppose that Φ is satisfiable with model $\mathcal{M}$ and that K out of the $N-1$ disequalities are satisfied by $\mathcal{M}$. The sub-tree that covers the previous class is refined into a tree that covers the new classes. We claim that we require at most $2 \cdot (\log N + K)$ fresh literals for the sub-tree. To see this, consider the sequence (1) where K out of the $N-1$ literals are true and the rest are false. If there are more than two contiguous literals that are false, then two neighbors must be summarized by an internal literal (a literal ϕ_i). We will reuse this summary when rebuilding the tree for the new constraints. The number of literals it takes to cover a sequence of false literals is therefore at most $\log N$. So we can build a tree of size $\log N + K$ above these together with the K fresh literals in the leaves. The total number of fresh literals required to cover the new equivalence classes is therefore $2 \cdot (\log N + K)$ (half for the leaves and half for the internal nodes). We also have to ensure that the upper and lower-level trees are balanced so that we can update at most $O(\log N)$ literals from the root to the leaves when updating the partitions. So

consider a set of balanced trees of different heights. We extract an *almost balanced* tree by eagerly joining any two trees of the same height until all trees have different heights. Then create an almost balanced tree out of the remaining trees. The maximal depth of the resulting two-level tree is then at most $\log N + 1$. By *almost balanced* we mean a binary tree (example on the right) that satisfies the following predicate $AB(n)$, defined recursively over binary trees for nodes n: $AB(n)$ holds if n is a leaf, or n is balanced, or if $n.right$ is balanced, $height(n.left) < height(n.right)$ and $AB(n.left)$.



The construction ensures that the number of fresh literals introduced is at most $O(N \log N)$. To justify this, let N_0 be the original number of terms. During one iteration we introduce M_j new classes. The new classes have sizes $a_1 N, \ldots, a_{M_j} N$, where $\sum_{i=1}^{M_j} a_i = 1$ and $\frac{N-1}{N} \geq a_1 \geq a_2 \geq \cdots \geq a_{M_j} \geq \frac{1}{N}$, so that a_1 is the fraction in the largest class. Thus, K is proportional to $(1 - a_1)N$; the contribution $\log N$ from the false literals is dominated by N. The cost of partitioning a class with N ($N \leq N_0$) terms is bounded by

$$T(N, N_0) \leq O(M_j \log N_0) + O((1 - a_1)N) + \textstyle\sum_{i=1}^{M_j} T(a_i N, N_0), \quad N > 1$$

where $T(1, N_0) = O(1)$, $O(M_j \log N_0)$ is the cost of updating the upper layer tree, $O((1 - a_1)N)$ is the number of literals introduced for updating the lower level tree for the current class, and $T(a_i N, N_0)$ is the number of fresh literals used for further refining the new classes. Since there can be at most N classes, we have $\sum_j M_j \leq N$. We claim that the recurrence $T(N, N_0)$ is bounded by $O(N \log N_0)$, so $T(N_0, N_0)$ is bounded by $O(N \log N)$. To verify this, first separate the contribution $O(M_j \log N_0)$ from T. The contribution expands into a sum $\sum_j O(M_j \log N_0)$, which by the bound on $\sum_j M_j$, is $O(N \log N_0)$. Thus,

$$T(N, N_0) \leq O(N \log N_0) + T'(N), \text{ where } T'(N) \leq O((1-a_1)N) + \textstyle\sum_{i=1}^{M_j} T'(a_i N).$$

We can over-approximate the cost $\sum_{i=2}^{M_j} T'(a_i N)$ by $T'((1 - a_1)N)$ (as the cost of the latter includes splitting a class of size $(1 - a_1)N$ into smaller classes), so the bound to analyze is: $T'(N) \leq O((1-a_1)N) + T'(a_1 N) + T'((1-a_1)N)$. The upper bound for the size contribution a_1 of the largest class decreases in each unfolding because $\frac{M-1}{M} < \frac{N-1}{N}$ for $M < N$, so we will just assume that it remains fixed at a_1. Thus, the depth of unfolding T' is bounded by $-\log N / \log a_1$, and the contribution of $O((1 - a_1)N)$ in each level adds up to $(1 - a_1)$, so the overall cost is the product $O(N \log N)$.

Space-Optimized Partition Refinement via Term Sharing (HSOPR) Similar to the hybrid algorithm between BPR and IPR, we experimented with an algorithm that asserts a disjunction of N disequations at each refinement iteration, but uses chains rather than stars of disequations following (1). This yields an algorithm with $O(N^2)$ space complexity as there are $O(N)$ iterations, each of which creates a disjunction of size N. However, at most $O(N \log N)$ of the disequalities are fresh, resulting in a high degree of incrementality.

4 Practicalities

There are several important details that are significant in an implementation of these algorithms. We will describe the most prolific ones here that we encountered in the context of Z3. We believe these are generic issues.

Canonicity The model-based algorithms all rely on a solver providing models with the ability to evaluate terms. The requirement is that if two terms t_1, t_2 are the same under an interpretation, then the evaluation under a model $\mathcal{M}$ is the same: $\mathcal{M}(t_1) = \mathcal{M}(t_2)$. We say that the interpretations are *canonizing* for a sort. Z3 produces canonizing interpretations for sorts Booleans, bit-vectors, integers, reals (for linear constraints), and algebraic data types that use sorts with canonizing interpretations. An example algebraic data type that is canonizing is the sort of finite lists of integers. Another example is finite lists of finite lists of integers. Terms of sort finite lists over arrays are on the other hand not canonizing. The implementation falls back to a version of basic partition merging for terms of non-canonizing sorts.

Array Values Z3 does not produce canonizing interpretations for arrays. So if we are given terms t_1, t_2, t_3 whose sorts are (one-dimensional) arrays, Z3's evaluation of these terms under $\mathcal{M}$ does not produce canonical values. There is a simple trick, however, that takes care of arrays in many cases: Due to extensional equality of arrays, the equality partition for $\{t_1, t_2, t_3\}$ under Φ is the same as the equality partition for $\{t_1[i], t_2[i], t_3[i]\}$ under Φ, where i is a fresh index variable.

Pre-partitioning Based on Sorts SLAYER queries for partitions of several sorts of terms at the same time. We found that the merging-based algorithms benefited significantly from pre-partitioning the terms by sort. It was particularly important to distinguish terms in the image of the translation of array terms above from those that genuinely have the same sort as the range of the array. In such cases, not distinguishing by sorts leads to logically more difficult problems. The implementations of the refinement algorithms include an optimization where the initial partition is not taken to be the coarsest one, but is computed from the model generated by the first satisfiability check. This first model will yield an initial partition which distinguishes all terms of distinct sorts, except those produced by the array translation, which the implementation explicitly separates from the others.

Knowing the Terms Z3 can provide an evaluator given a model $\mathcal{M} \models \Phi$ that evaluates subterms in Φ. To force all terms $t_1, \ldots, t_N$ to be in Φ, we initialize Φ to $\Phi \wedge K(t_1) \wedge \cdots \wedge K(t_N)$, where K is a fresh predicate (*Known*).

Diversity All algorithms that rely on models require fewer iterations if the models are as *diverse* as possible. For example, if Φ is consistent with all $t_1, \ldots, t_N$ evaluating to different values, we are done in a single iteration. But Z3 is not required to produce diverse models. In the case of algebraic data types Z3 searches for models by building small instances. So for lists, Z3 always attempts to set a term to *nil* (the empty list). For arithmetic, Z3 supports a configuration, `arith.random_initial_value=true`, for shaking up initial values. Otherwise values of variables default to 0.

5 Empirical Evaluation

We used SLAYER running on device driver benchmarks to evaluate the algorithms. Statistics were gathered as SLAYER was running, to accurately reflect the actual mode of usage, which is through Z3's incremental programmatic interface. No individual implied equalities queries exhausted time or memory resources, although there are fewer queries for some algorithms in cases where the client analyzer exhausted resources. SMT-LIB2 benchmark files for most queries were generated during separate runs and are available online.[1] Appendix B contains many more details.

Figs. 1–6 each compare two algorithms. Results are reported only for the instances where the formula is satisfiable, all the algorithms behave equivalently with inconsistent formulas and so those points only add clutter. The right y-axis and top x-axis are the run times for the two algorithms. Times are reported in seconds, where query times measured below 50ms have been reported as 50ms since such instances are uninterestingly easy and accurately measuring such short times is problematic. Instances that are quickly solved by only one algorithm appear on an axis. A solid $y = x$ line is shown, as well as dotted lines indicating speedup and slowdown factors of $10x$, $100x$, and so on. Each plot includes a solid trend line that has been fit to the data, for what it is worth given the very high degree of variation and delicacy of nonlinear fitting. Each plot also includes two lines from upper-left to lower-right, associated with the left y-axis and bottom x-axis. The solid line indicates the number of instances where the algorithm on the right y-axis was faster than the algorithm on the top x-axis by at least the left y-coordinate seconds, and vice versa for the dashed line. The key reports the area under these curves, representing the cumulative speedups.

Fig. 1 compares the run times of the naïve Basic Partition Merging and semi-naïve Model-based Partition Merging algorithms. The conclusion is extremely clear-cut: despite the fact that the algorithms have the same theoretical complexity, in virtually all cases the model-based algorithm shows 10–5000x speedups.

Fig. 2 compares the run times of the MPM and new Incremental Partition Refinement algorithms. Here the results are still very clear-cut, though not as dramatic as with the comparison to the most basic algorithm. There is a scattering of, generally easier, instances where MPM outperforms IPR, but the bulk of the harder instances see between 10–100x speedups with IPR.

The assumption-based algorithm, ABIPR, does not offer dramatic benefits over IPR. Fig. 3 shows that while ABIPR is slightly faster overall, and is trending to scale slightly better on the harder instances, there are many instances on which IPR is faster. Fig. 4 compares BPR to ABIPR, showing that for our evaluation suite manipulating the proposition heap results in a significant overhead. ABIPR is faster on the easier instances, but BPR has larger speedups. So while on our benchmark suite the overall time spent by BPR is slightly higher than ABIPR, there is a slight trend toward BPR scaling better.

Fig. 5 compares HIPR versus BPR, showing that the hybrid incremental algorithm is overall somewhat faster than the basic version.

[1] http://research.microsoft.com/apps/pubs/default.aspx?id=215371

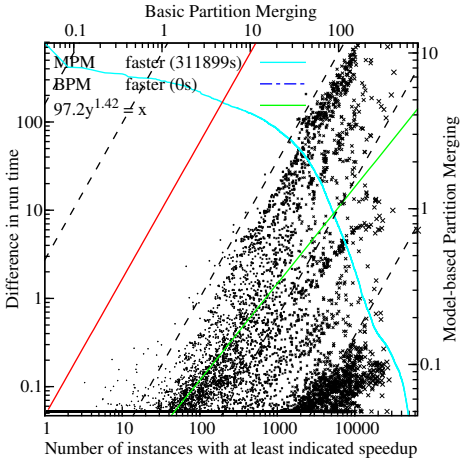


Fig. 1. Run time of MPM vs BPM

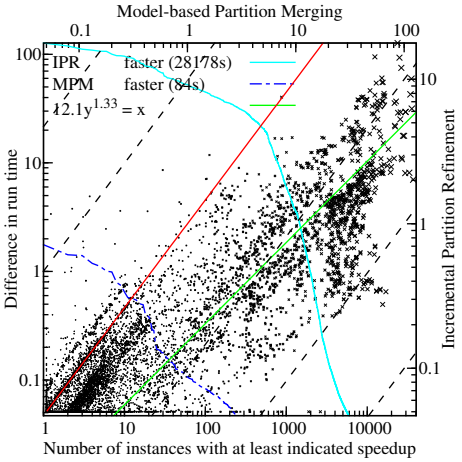


Fig. 2. Run time of IPR vs MPM

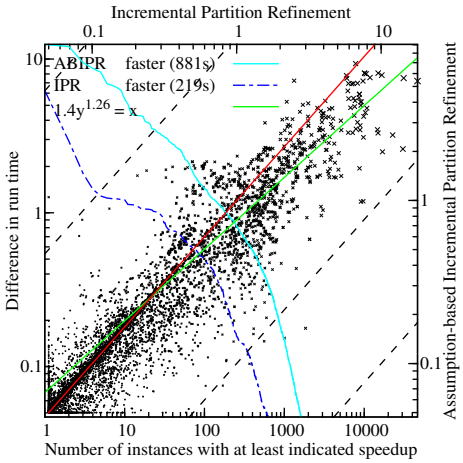


Fig. 3. Run time of ABIPR vs IPR

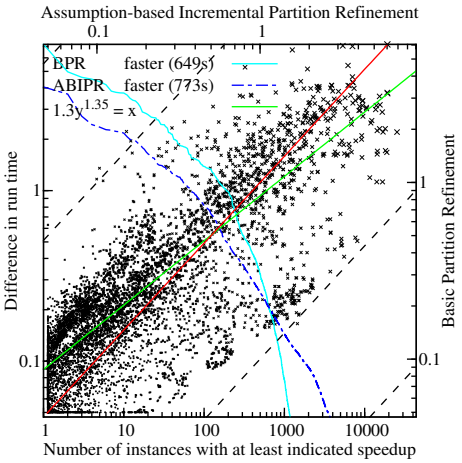


Fig. 4. Run time of BPR vs ABIPR

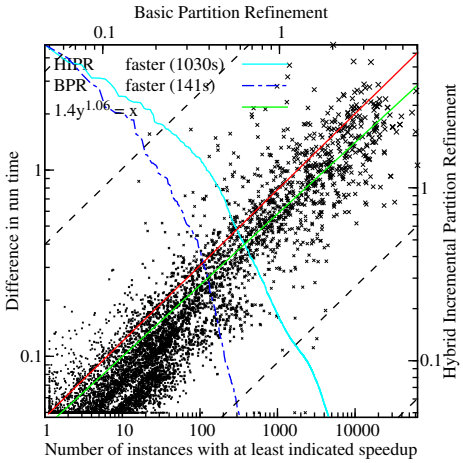


Fig. 5. Run time of HIPR vs BPR

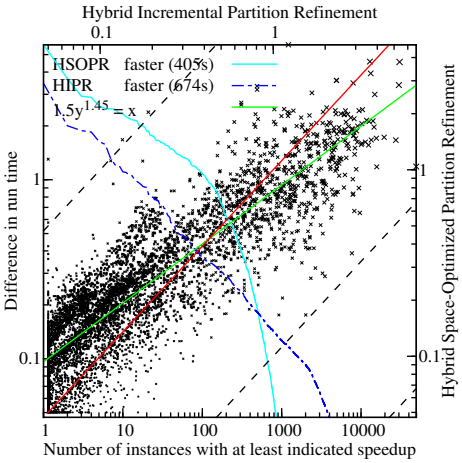


Fig. 6. Run time of HSOPR vs HIPR

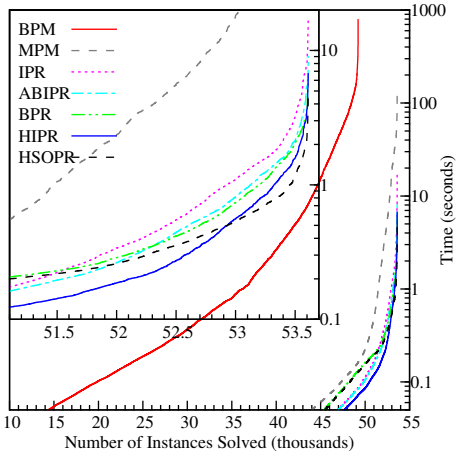


Fig. 7. Run time vs No. instances solved

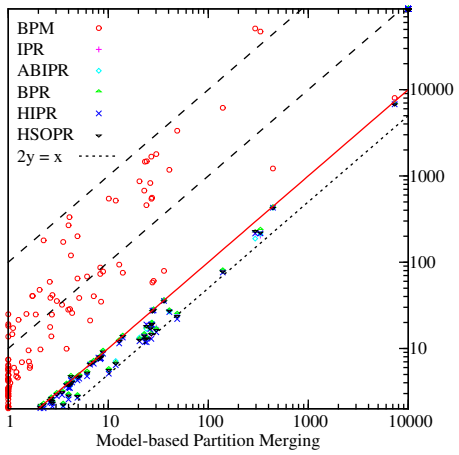


Fig. 8. Analysis time relative to MPM

Fig. 6 shows that the HSOPR algorithm based on the $O(N \log N)$-space algorithm has some overhead relative to HIPR leading to slower performance on the easier instances, but scales better on the harder instances.

To provide an overall picture of all the algorithms discussed, Fig. 7 shows the number of instances solved within a given run time. From this we see that:

- BPM is much slower than the others.
- MPM is much faster than BPM but still significantly slower than the others.
- HIPR is fastest on easy instances, but is overtaken by HSOPR as it scales better.
- BPR and HSOPR are slower on easier instances, but scale better than IPR and ABIPR, eventually overtaking them.

The implementations of BPR and HSOPR are similar. They require a little more work outside of the SMT solver than the IPR, ABIPR and especially HIPR.

Fig. 8 compares the overall SLAYER analysis run times using each algorithm relative to using MPM. The results show that, while computing implied equalities is only one sub-algorithm, improving it still yields considerable speedups of 10–100x or more over BPM and an additional 2x over MPM. SLAYER hardly works with BPM, computing implied equalities is *the* bottleneck. With MPM, computing implied equalities is no longer the only bottleneck, but a significant speedup is still achieved by the refinement-based algorithms. The differences between the various refinement-based algorithms are not as apparent on full analysis runs, though HIPR is most often fastest, and ABIPR and IPR have some notable wins on hard instances.

In summary, the model-based algorithms dramatically outperform BPM. Among the model-based algorithms, partition refinement is clearly superior to the partition merging done by MPM. Between the partition refinement algorithms, relying on sub-formula sharing to achieve incrementality is at least as effective as using the proposition heap. And finally, the additional reuse enabled by following the $O(N \log N)$-space algorithm results in noticeably-better scaling.

6 Conclusions

Prompted by benchmarks from an application in program analysis, we developed efficient algorithms for inferring implied equalities. It generalizes congruence closure along two dimensions: from conjunctions of equations to general Boolean formulas, and from the free theory of uninterpreted functions to the variety of theories the employed solver supports. To our knowledge, only initial basic algorithms had been previously proposed. The overall result is a drastic reduction in solver calls over simple algorithms. An empirical evaluation using non-synthetic, but single-source and generally short-running, benchmarks shows speedups exceeding 10–100x over the implementation of Model-based Partition Merging previously available in Z3, which is already almost never less than 10x, and up to 5000x, faster than the basic algorithm.

References

1. Berdine, J., Cook, B., Ishtiaq, S.: SLAyer: Memory safety for systems-level code. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV. LNCS, vol. 6806, pp. 178–183. Springer (2011)
2. Berman, C., Trevillyan, L.: Functional comparison of logic designs for VLSI circuits. In: Kannan, L.N. (ed.) ICCAD. pp. 456–459. IEEE Computer Society (1989)
3. Brand, D.: Verification of large synthesized designs. In: Lightner, M.R., Jess, J.A.G. (eds.) ICCAD. pp. 534–537. IEEE Computer Society (1993)
4. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
5. Downey, P.J., Sethi, R., Tarjan, R.E.: Variations on the common subexpression problem. J. ACM 27(4), 758–771 (1980)
6. Heule, M., Biere, A.: Blocked clause decomposition. In: McMillan, K.L., Middeldorp, A., Voronkov, A. (eds.) LPAR. LNCS, vol. 8312, pp. 423–438. Springer (2013)
7. Hoder, K., Voronkov, A.: Comparing unification algorithms in first-order theorem proving. In: Mertsching, B., Hund, M., Aziz, M.Z. (eds.) KI. LNCS, vol. 5803, pp. 435–443. Springer (2009)
8. Lev-Ami, T., Sagiv, S.: TVLA: A system for implementing static analyses. In: Palsberg, J. (ed.) SAS. LNCS, vol. 1824, pp. 280–301. Springer (2000)
9. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
10. Nieuwenhuis, R., Oliveras, A.: Fast congruence closure and extensions. Inf. Comput. 205(4), 557–580 (2007)
11. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM 53(6), 937–977 (2006)
12. Paterson, M., Wegman, M.N.: Linear unification. J. Comput. Syst. Sci. 16(2), 158–167 (1978)
13. Robinson, J.A.: Computational logic: The unification computation. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence 6, pp. 63–72. Edinburgh University Press (1971)
14. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
15. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. ACM 22(2), 215–225 (1975)

A Additional Details of Equality Consequence Algorithms

For reference and to clarify the experimental comparisons, we present the algorithms in more detail.

A.1 Basic Partition Merging (BPM)

Algorithm 2 implements the elementary approach Basic Partition Merging for finding all implied equalities. For every pair of terms t_i, t_j, it uses an SMT solver query to check if the equality between t_i and t_j is forced. If t_i and t_j are equal, the partition P is updated by merging the classes where i and j occur. It uses the routines *find* and *union* from Section 2.

Algorithm 2: Basic Partition Merging (BPM)

Input: formula Φ and set of terms $t_1, \ldots, t_N$
Output: equality partition P of the set $\{1, \ldots, N\}$

```
1 P ← {{1}, ..., {N}};
2 foreach index i = 1 ... N do
3     foreach index j = i + 1 ... N do
4         if find(P, i) ≠ find(P, j) ∧ CheckValid(Φ → t_i ≃ t_j) then
5             P ← union(P, i, j);
6 return P;
```

Theorem 1 *Algorithm 2 computes the equality partition of Φ for terms $t_1, \ldots, t_N$. It requires $O(N)$ space and $O(N^2)$ solver calls.*

Proof (sketch) Correctness of Algorithm 2 follows immediately from the assumed correctness of the SMT solver. A complexity of $O(N^2)$ solver queries is also immediate.□

A.2 Model-based Partition Merging (MPM)

Suppose that Φ is satisfiable with an interpretation $\mathcal{M}$ that makes terms t_i and t_j different. It is then not necessary to check if the equality between t_i and t_j is forced. Algorithm 3 implements the Model-based Partition Merging approach, using models as a filter for which equalities to check.

Theorem 2 *Algorithm 3 computes the equality partition of Φ for terms $t_1, \ldots, t_N$. It requires $O(N)$ space and $O(N^2)$ solver calls.*

Proof (sketch) Correctness of Algorithm 3 follows directly from that of Algorithm 2 and the fact that only equalities true in $\mathcal{M}$ can be true in all interpretations. The $O(N^2)$ theoretical complexity is unchanged. □

Algorithm 3: Model-based Partition Merging (MPM)

Input: formula Φ and set of terms $t_1, \ldots, t_N$
Output: equality partition P of the set $\{1, \ldots, N\}$

```
1  if Φ is unsat then
2      return {{1, ..., N}};
3  M ← interpretation satisfying Φ;
4  Q ← ∅;
5  foreach index i = 1 ... N do
6      v ← M(t_i);
7      if v ∉ dom Q then
8          Q[v] ← ∅;
9      Q[v] ← Q[v] ∪ {i};
10 P ← ⋃{Basic Partition Merging(Q[v]) | v ∈ dom Q};
11 return P;
```

A.3 Basic Partition Refinement (BPR)

Algorithm 4 is a basic algorithm based on partition refinement. It requires at most N satisfiability checks, but creates a new formula of size up to N in each iteration. While models were used as *hints* in the previous Algorithm 3, models are now an integral part of the algorithm: the models indicate where the partitions should be refined.

Theorem 3 *Algorithm 4 computes the equality partition of Φ for terms $t_1, \ldots, t_N$. It requires $O(N^2)$ space and $O(N)$ solver calls.*

Proof (sketch) If the input formula is unsatisfiable, then it vacuously implies every possible equality, so the initial maximally coarse partition is correct. Otherwise Φ is satisfiable and the first iteration of the outer loop establishes the invariant that all further, refinement, iterations maintain. During refinement, the partition acts as a conjecture that all equalities between terms in the same class are implied by the input formula, call it Φ_0. The partition P and formula Φ are updated in sync, maintaining the property that P contains all equations implied by Φ_0 and Φ is satisfied by the interpretations that satisfy Φ_0 but violate an equality contained in P.

First note that lines 4–12 set P' to the coarsest partition that distinguishes all terms that P does as well as those terms that are given distinct values by $\mathcal{M}$. This works by considering each class of P in turn and constructing a map from values v to sets of the indices of terms that evaluate to v in $\mathcal{M}$. The union of the ranges of these maps forms the new partition.

Since P initially contains all equations, any equation not in P' is not implied, as witnessed by $\mathcal{M}$. The update to Φ at line 14 adds a constraint that is satisfied by interpretations which violate at least one of the equalities conjectured by the updated partition P. Therefore the invariant is established.

Preservation of the invariant follows similar reasoning. The invariant ensures that $\mathcal{M}$ satisfies Φ_0, and hence does not violate any equalities implied by Φ_0. It also ensures that P contains all implied equalities, and hence lines 4–12 set P' to a partition that

Algorithm 4: Basic Partition Refinement (BPR)

Input: formula Φ and set of terms $t_1, \ldots, t_N$
Output: equality partition P of the set $\{1, \ldots, N\}$

1 $P \leftarrow \{\{1, \ldots, N\}\}$;
2 **while** Φ *is satisfiable* **do**
3 $\quad \mathcal{M} \leftarrow$ interpretation satisfying Φ;
4 $\quad P' \leftarrow \emptyset$;
5 $\quad$ **foreach** $q \in P$ **do**
6 $\quad\quad Q \leftarrow \emptyset$;
7 $\quad\quad$ **foreach** $i \in q$ **do**
8 $\quad\quad\quad v \leftarrow \mathcal{M}(t_i)$;
9 $\quad\quad\quad$ **if** $v \notin$ **dom** Q **then**
10 $\quad\quad\quad\quad Q[v] \leftarrow \emptyset$;
11 $\quad\quad\quad Q[v] \leftarrow Q[v] \cup \{i\}$;
12 $\quad\quad P' \leftarrow P' \cup \bigcup\{Q[v] \mid v \in$ **dom** $Q\}$;
13 $\quad P \leftarrow P'$;
14 $\quad \Phi \leftarrow \Phi \wedge \bigvee_{i \in 1 \ldots N} t_i \not\simeq t_{\mathit{find}(P,i)}$;
15 **return** P;

also includes all implied equalities. The update at line 14 again brings Φ back in sync with P. Note that since P is refined monotonically, the latest constraint added to Φ is stronger than all previous constraints added to Φ_0.

Since every iteration breaks at least one equality, and there are N equalities in total, termination is guaranteed after at most N satisfiability checks.

When Φ is eventually found to be unsatisfiable, the invariant ensures that P contains all the implied equalities, and also that there is no way to violate any of the equalities in P. That is, P contains exactly the implied equalities. □

A.4 Incremental Partition Refinement (IPR)

Theorem 4 *Algorithm 1 computes the equality partition of Φ for terms $t_1, \ldots, t_N$. It requires $O(N)$ solver calls and constructs $O(N^2)$ fresh literals.*

Proof (sketch) Line 1 initializes P to the maximally coarse partition, which is correct since if the input formula is unsatisfiable, then it vacuously implies every equality. Lines 2–4 assert constraints initializing the proposition heap such that, as discussed above, the refinement loop invariant below holds.

The partition acts as a conjecture that all equalities between terms in the same class are implied by the input formula, call it Φ_0. Each iteration of the refinement loop (lines 5–27) updates the partition P and formula Φ, maintaining the invariant that P contains all equations implied by Φ_0, and Φ is satisfied by the interpretations that satisfy Φ_0 but violate an equality contained in P.

Refinement continues so long as Φ is satisfiable. The invariant ensures that $\mathcal{M}$ extracted at line 6 satisfies Φ_0, and hence does not violate any equalities implied by Φ_0. After clearing the temporary map Q, executing lines 26–27 preserves the refinement loop invariant, as discussed above. When Φ is eventually found to be unsatisfiable, the invariant ensures that P contains all the implied equalities, and also that there is no way to violate any of the equalities in P. That is, P contains exactly the implied equalities.

Since every iteration breaks at least one equality, and there are N equalities in total, termination is guaranteed after at most N satisfiability checks. $O(N^2)$ is trivial, since at worst each of the N iterations reconstructs the entire heap, which has size $2N - 1$.□

B Additional Experimental Results

Fig. 9 is an enlarged version of Fig. 7. Fig. 10 is analogous, but reports the cumulative run time consumed by the algorithms to solve each number of instances.

Figs. 11–31 compare the run times of each pair of algorithms, including enlarged versions of Figs.1–6. Note that the size of the points is proportional to the square root of the distance from the origin, to enable more detail to be shown for the more dense easier problems without making the more sparse harder problems invisible.

Fig. 32 is an enlarged version of Fig. 8.

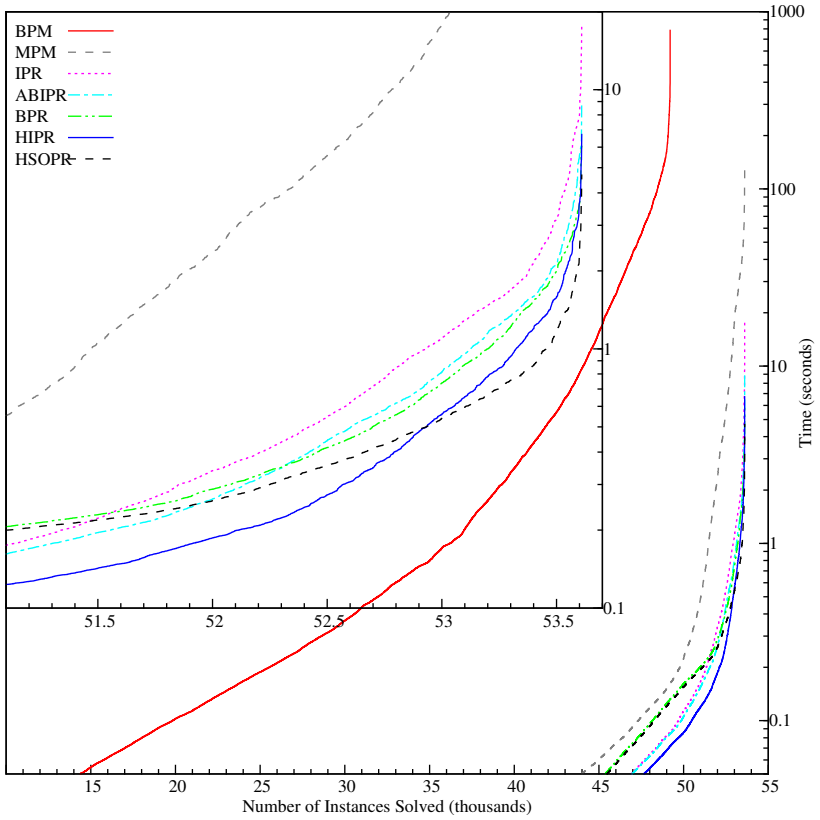


Fig. 9. Run time versus Number of instances solved

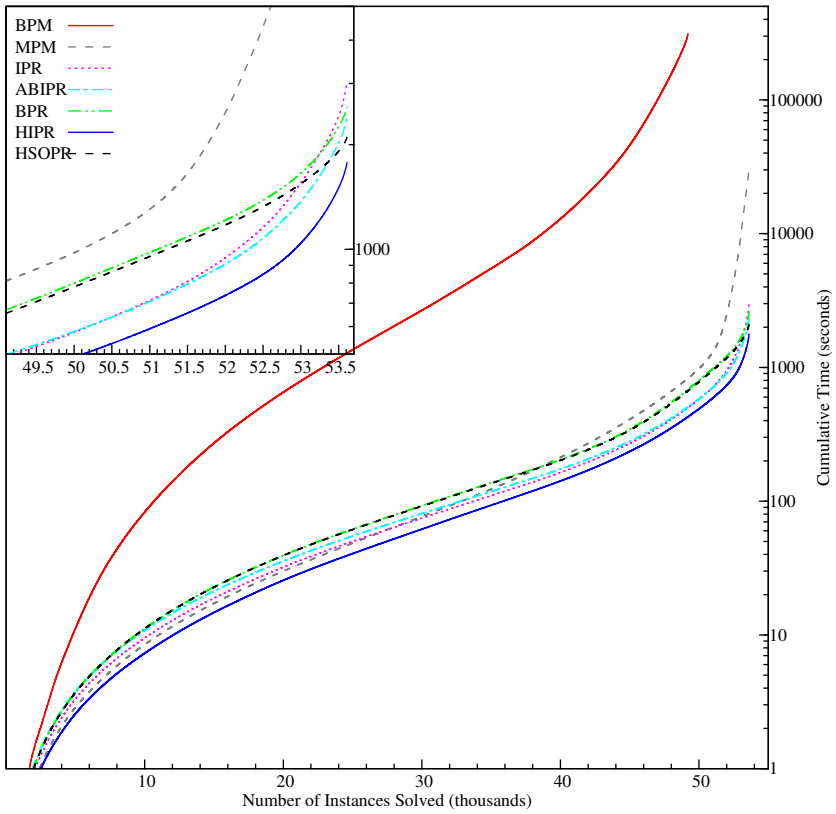


Fig. 10. Cumulative run time versus Number of instances solved

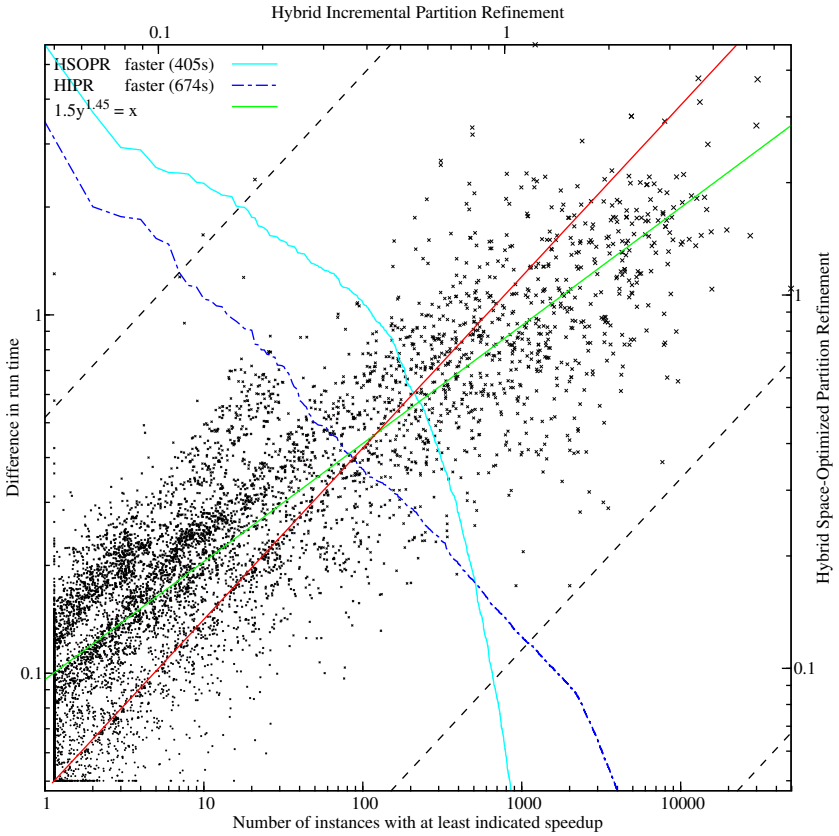


Fig. 11. Run time of HSOPR versus HIPR

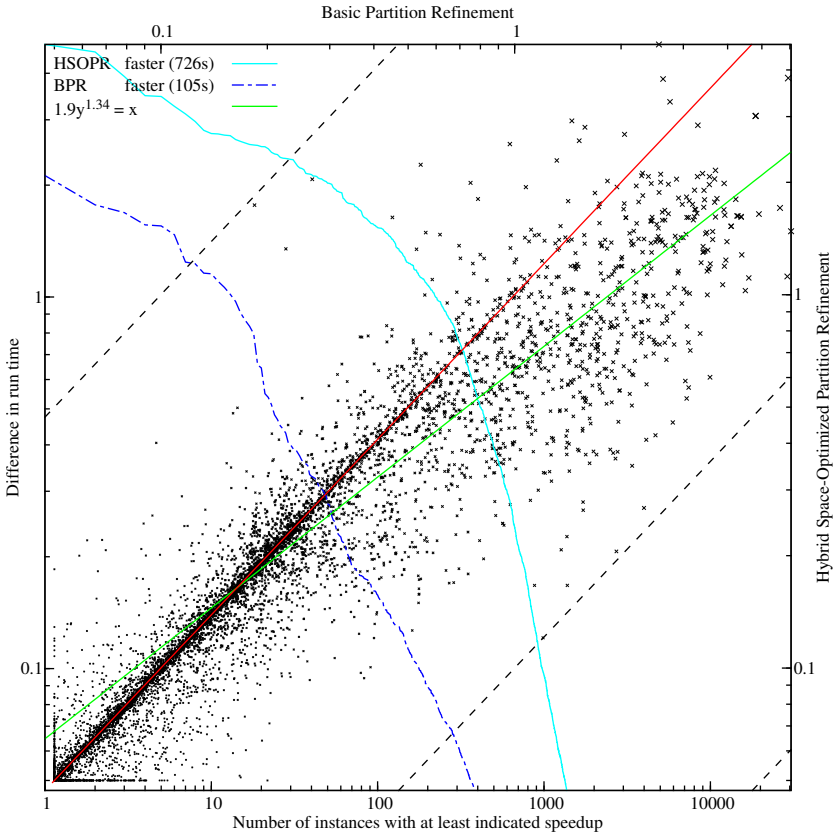


Fig. 12. Run time of HSOPR versus BPR

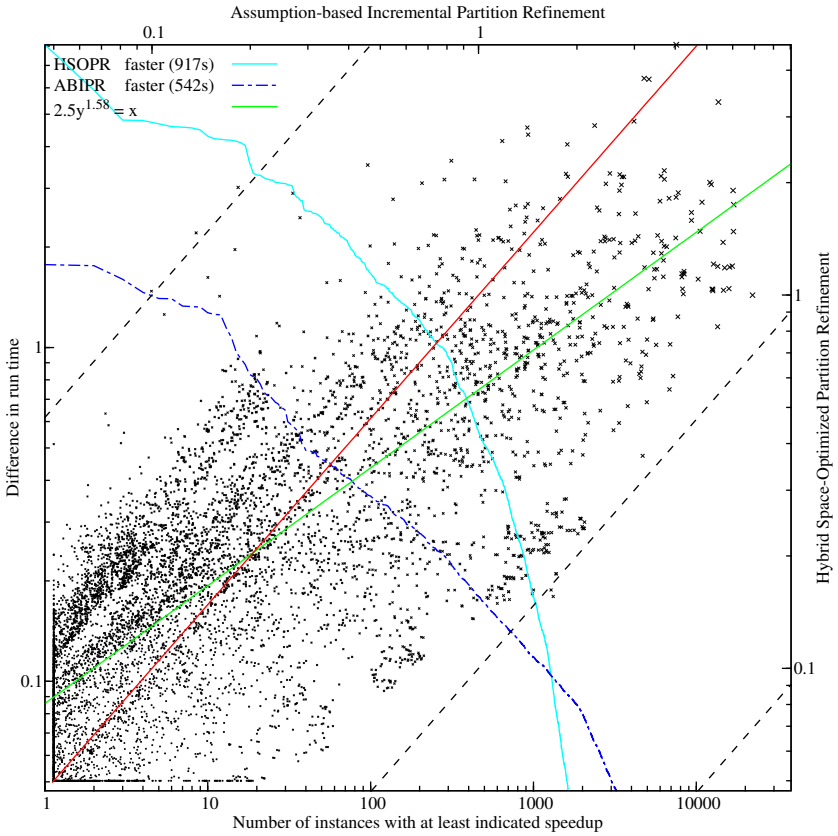


Fig. 13. Run time of HSOPR versus ABIPR

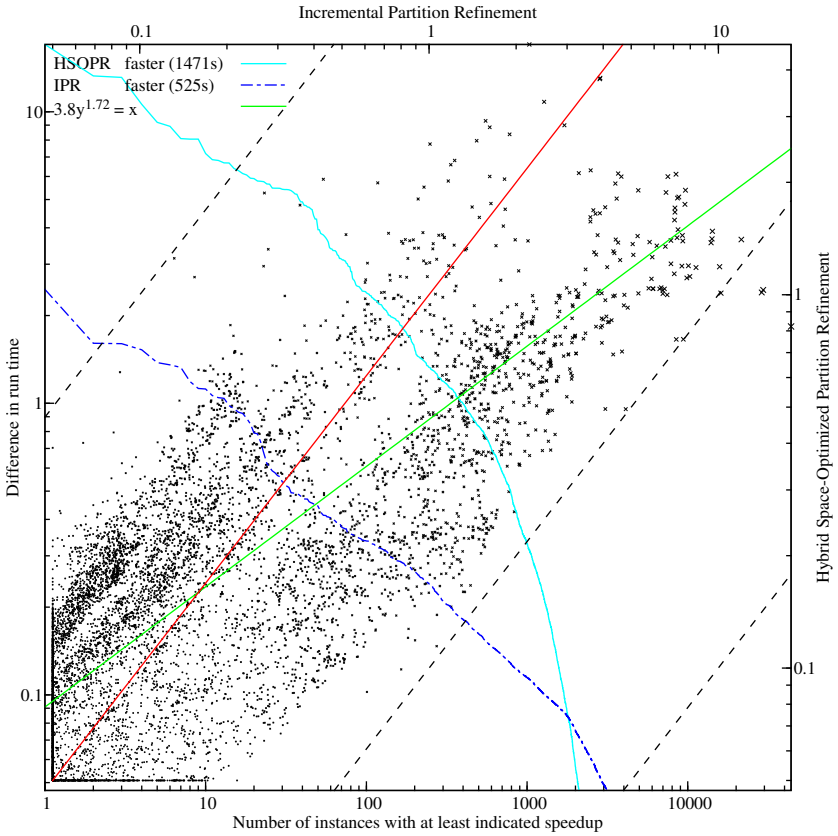


Fig. 14. Run time of HSOPR versus IPR

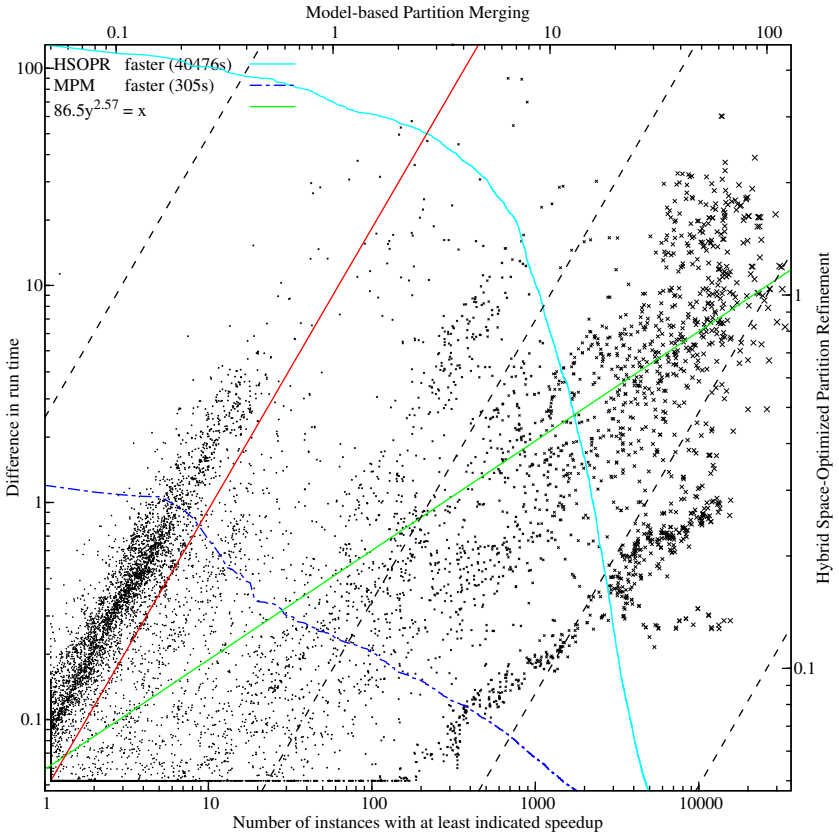


Fig. 15. Run time of HSOPR versus MPM

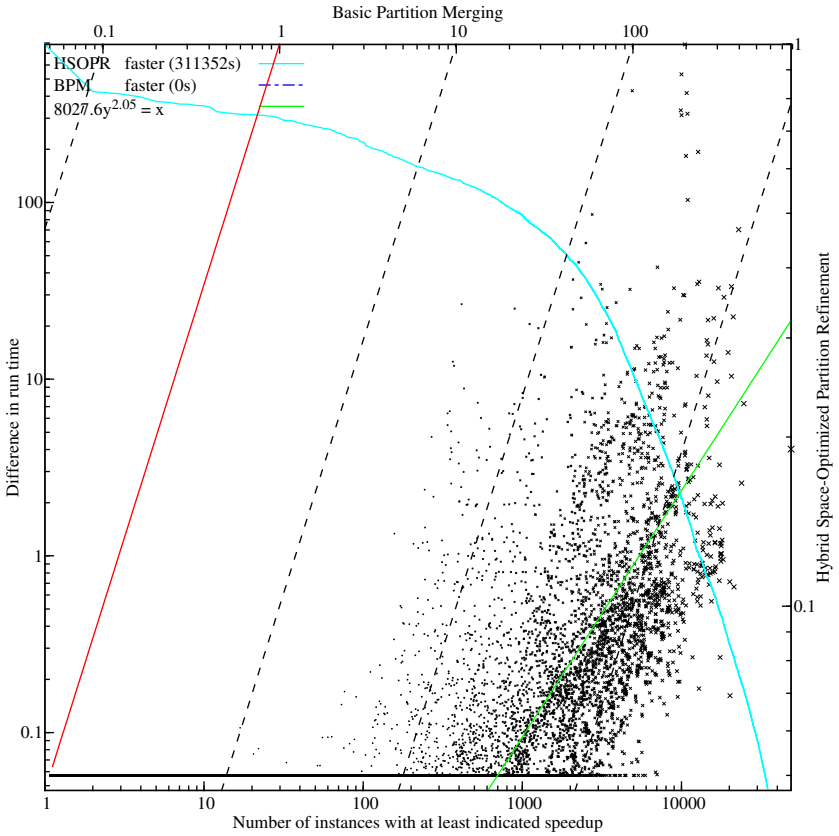


Fig. 16. Run time of HSOPR versus BPM

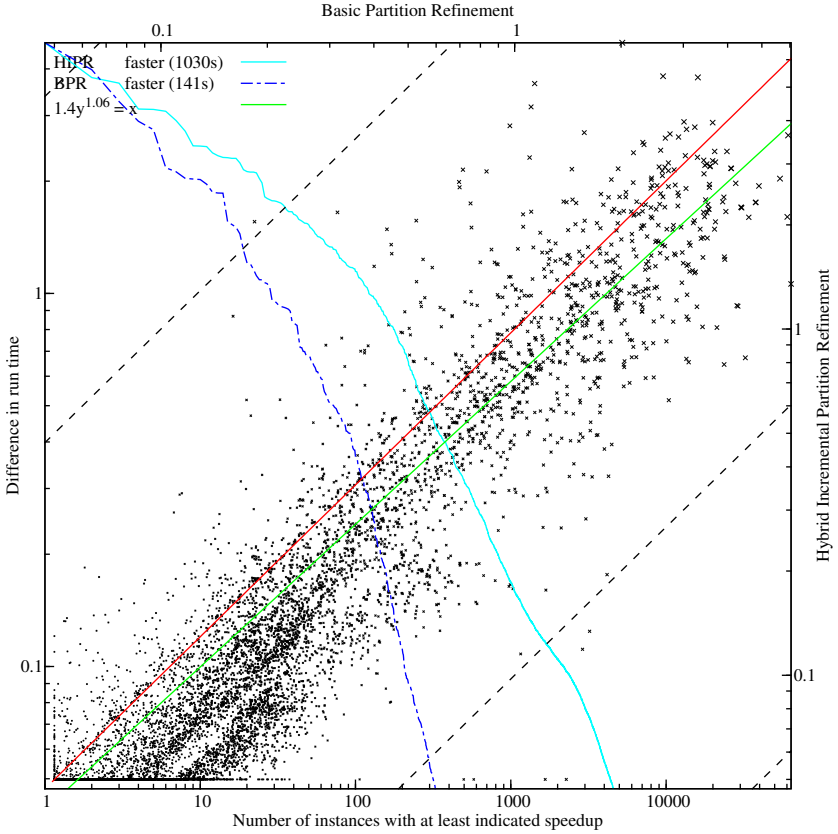


Fig. 17. Run time of HIPR versus BPR

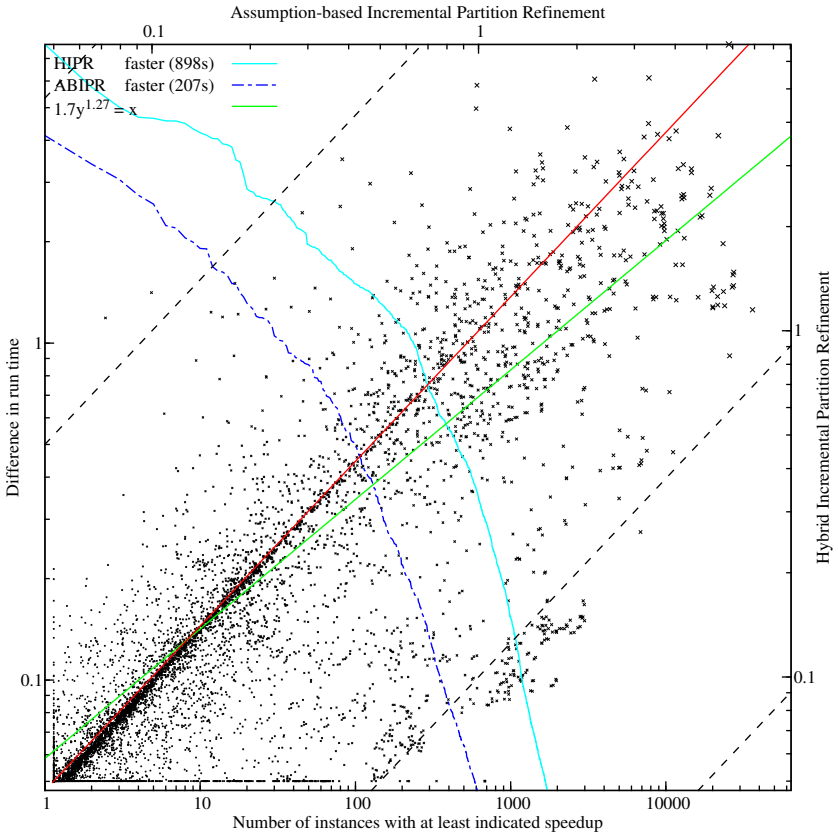


Fig. 18. Run time of HIPR versus ABIPR

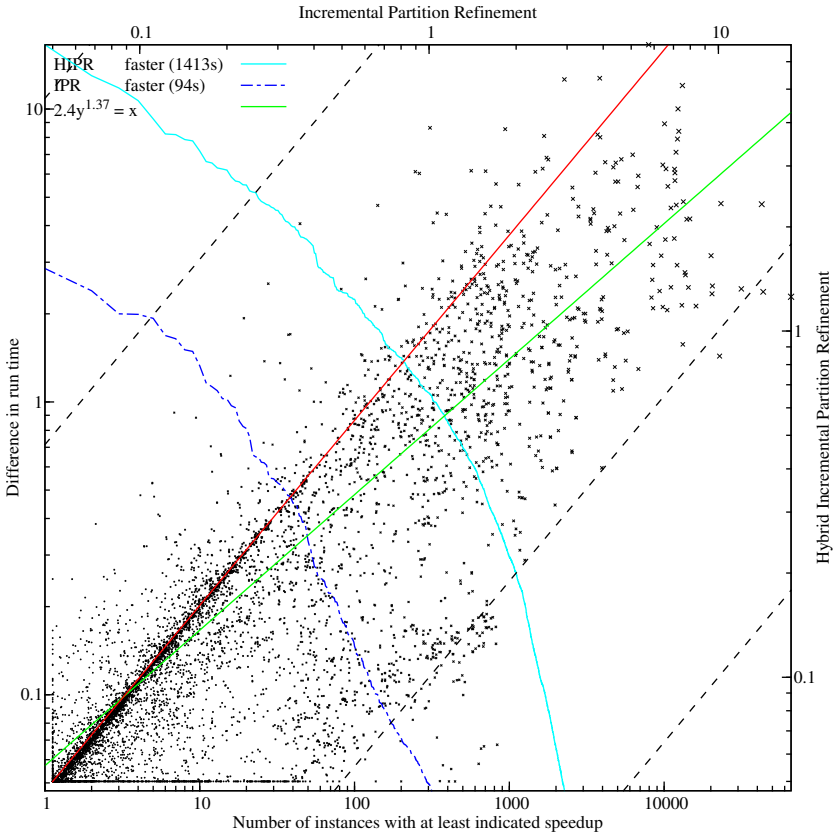


Fig. 19. Run time of HIPR versus IPR

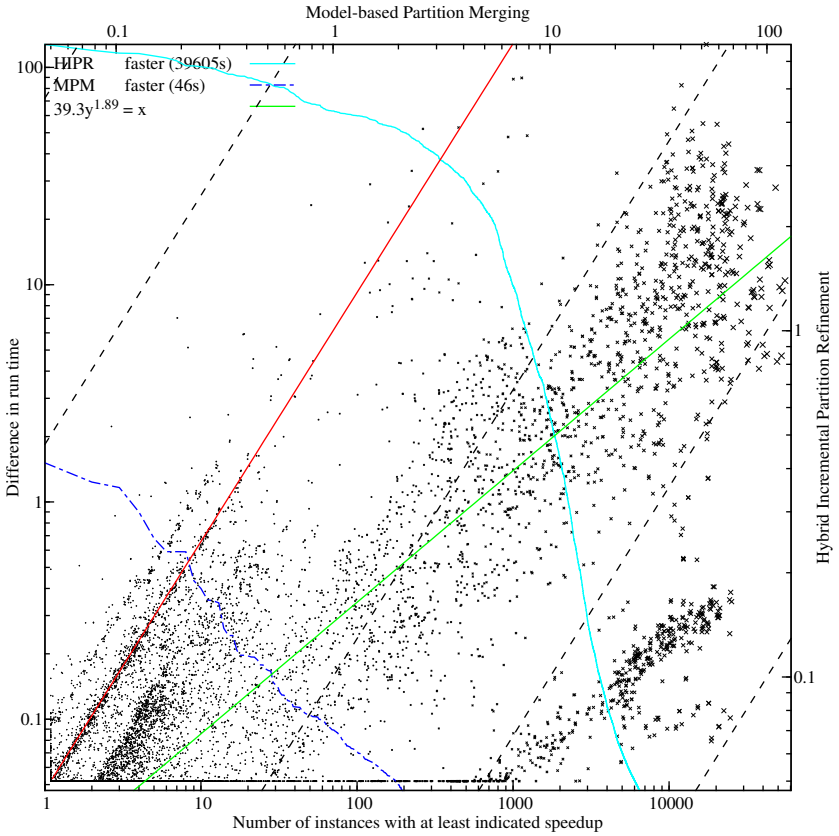


Fig. 20. Run time of HIPR versus MPM

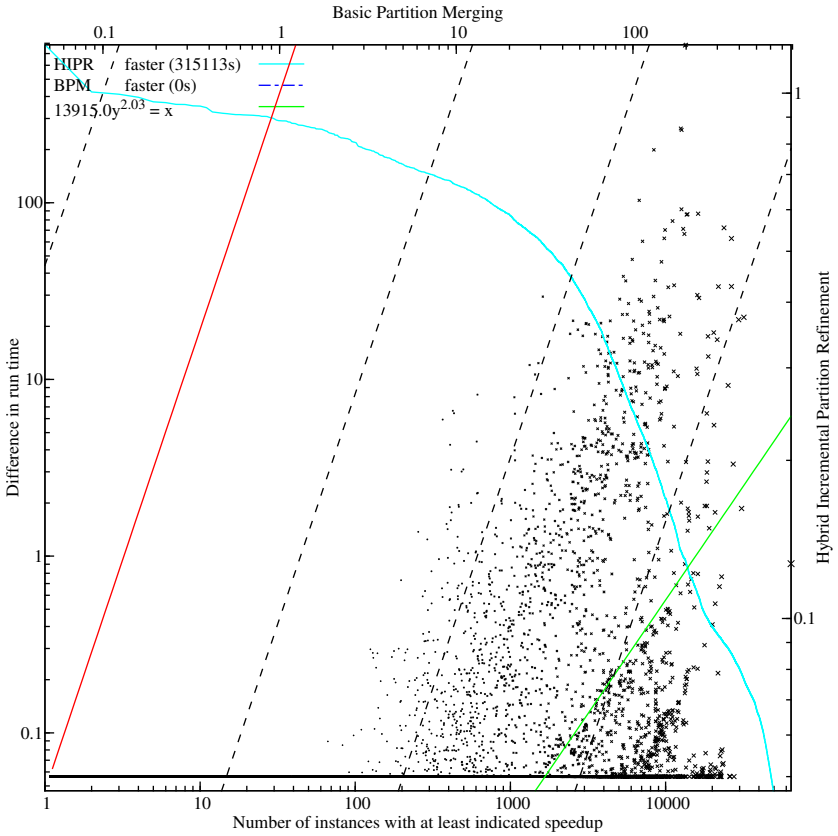


Fig. 21. Run time of HIPR versus BPM

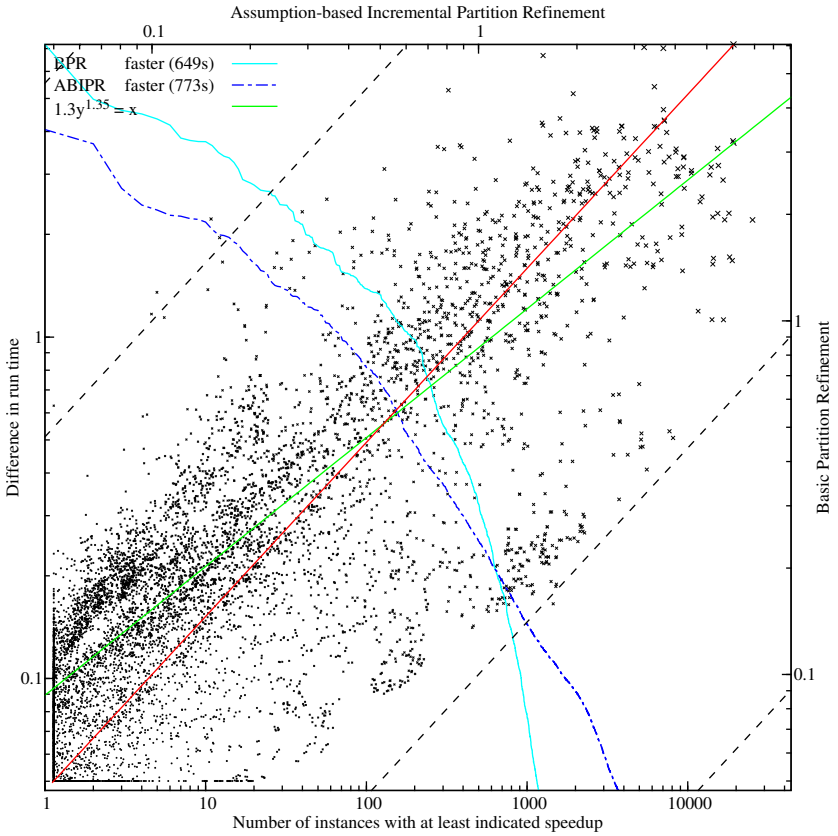


Fig. 22. Run time of BPR versus ABIPR

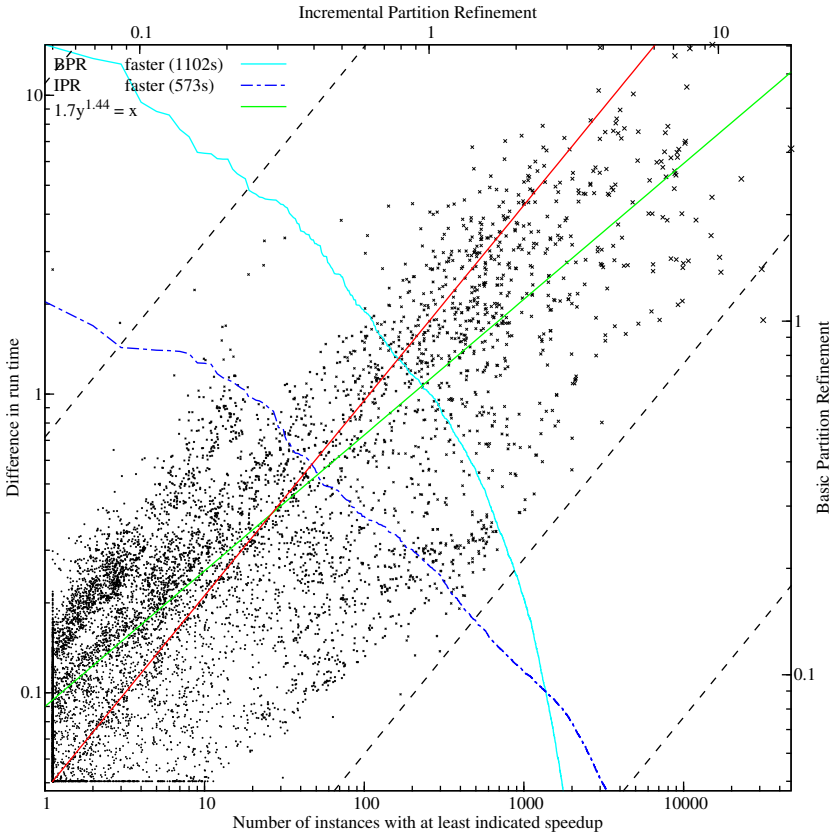


Fig. 23. Run time of BPR versus IPR

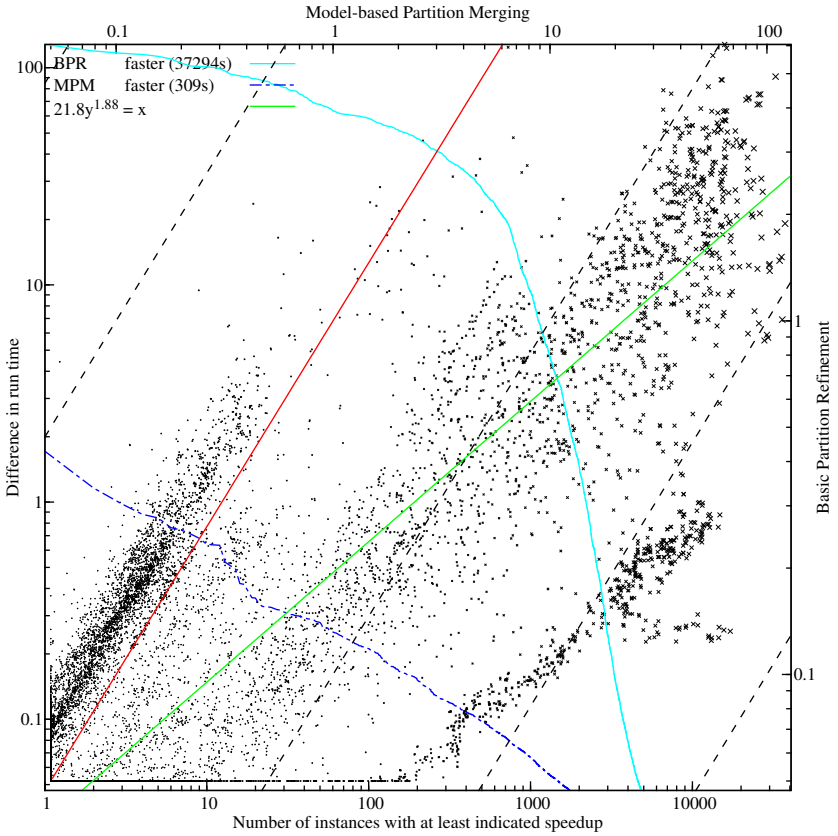


Fig. 24. Run time of BPR versus MPM

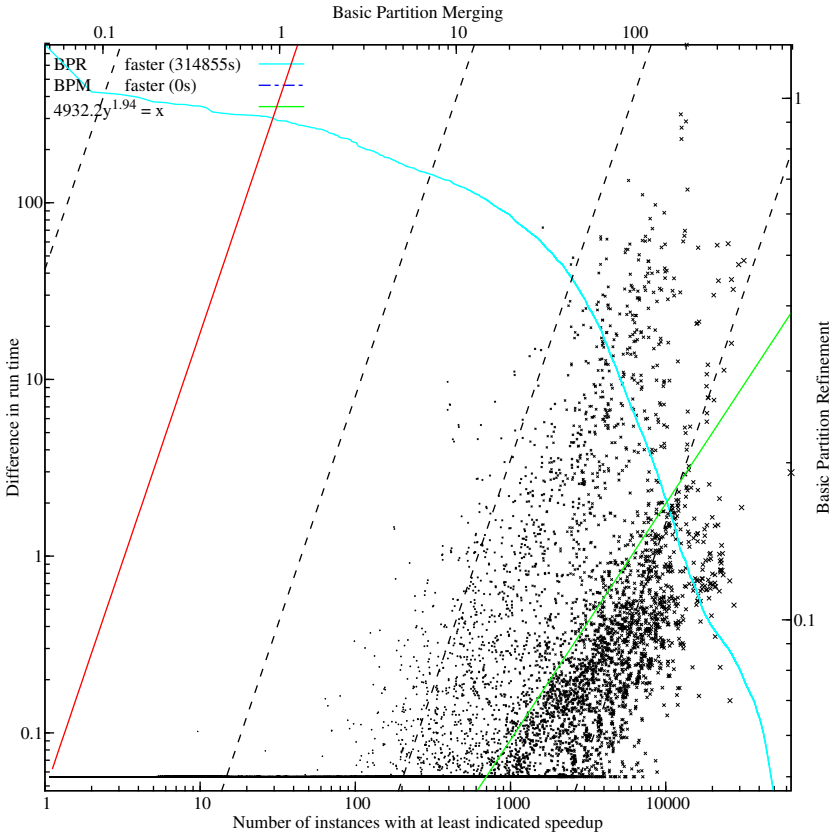


Fig. 25. Run time of BPR versus BPM

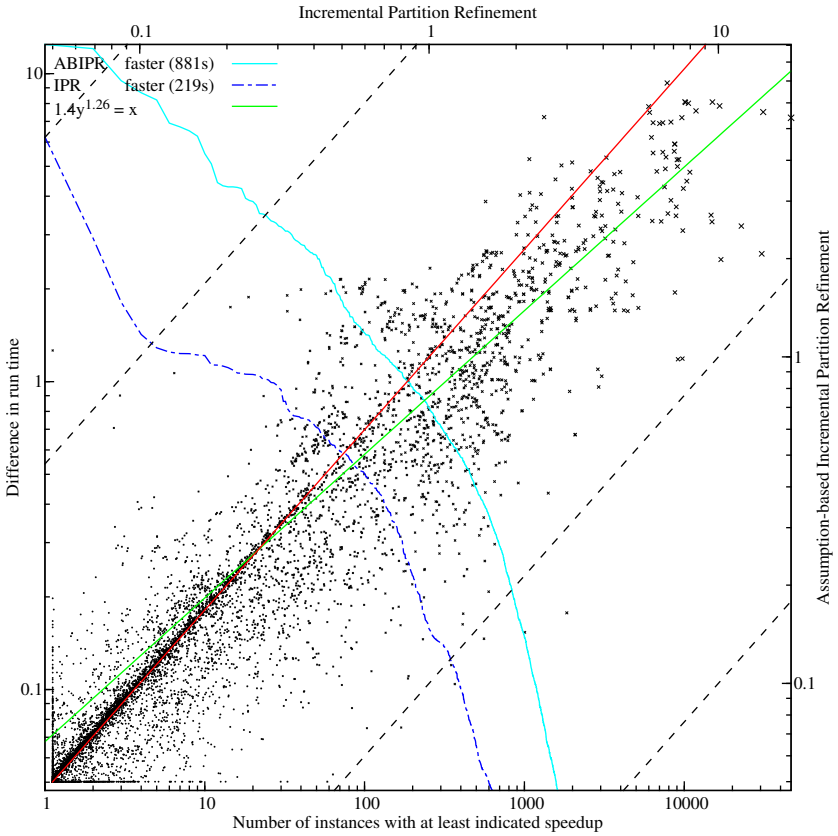


Fig. 26. Run time of ABIPR versus IPR

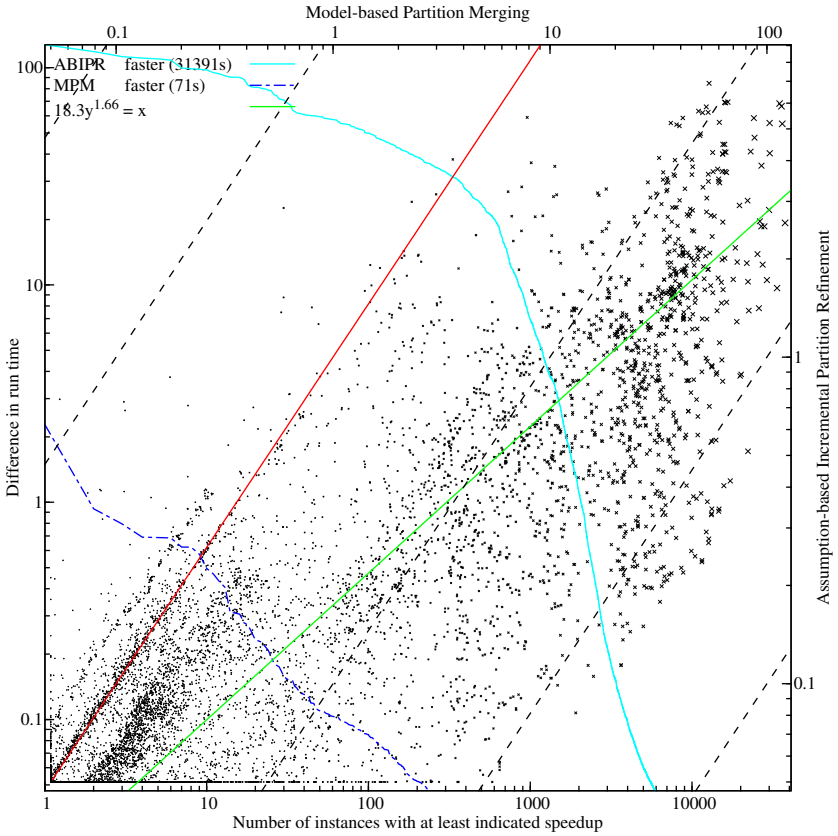


Fig. 27. Run time of ABIPR versus MPM

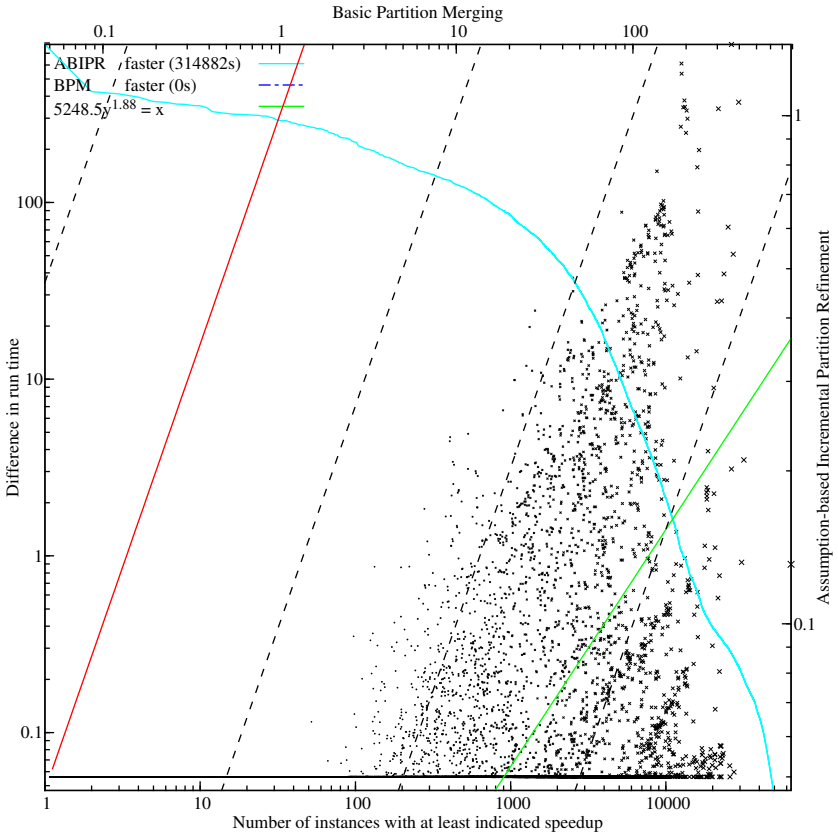


Fig. 28. Run time of ABIPR versus BPM

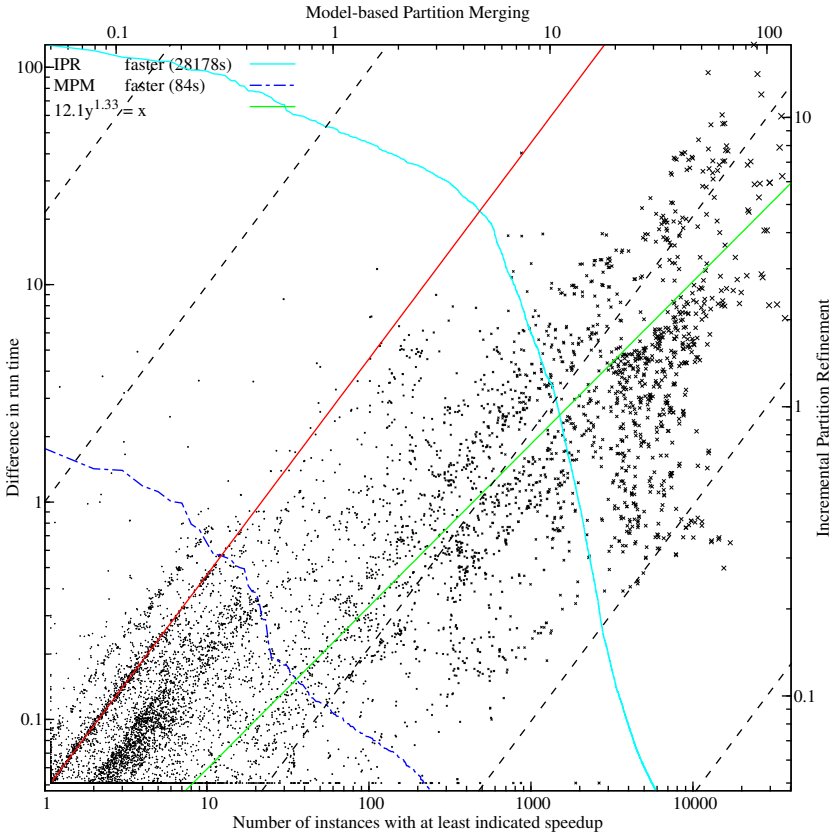


Fig. 29. Run time of IPR versus MPM

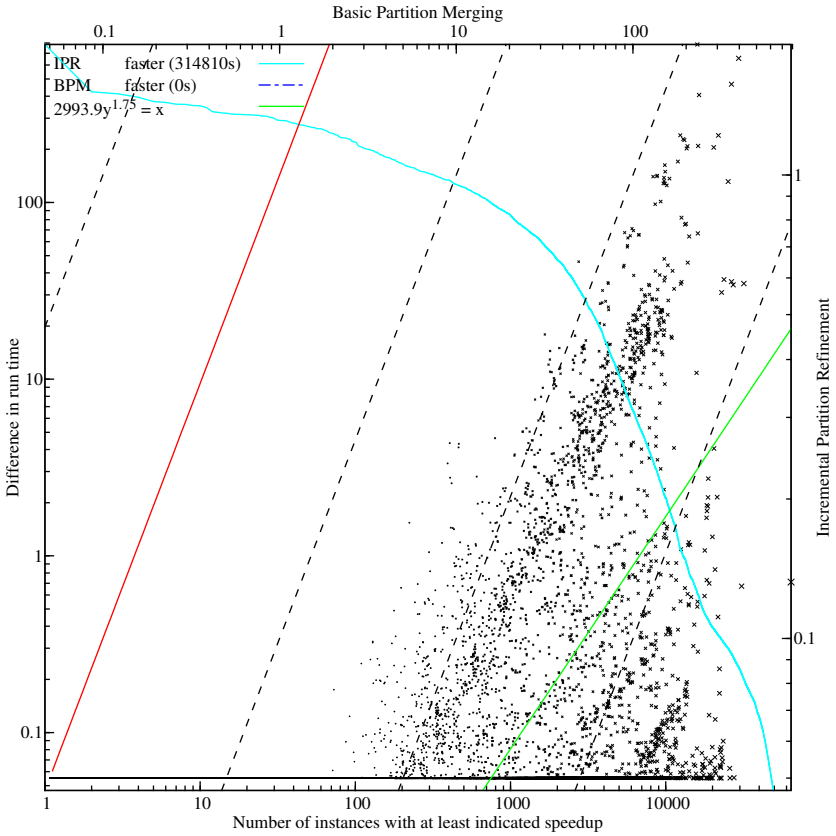


Fig. 30. Run time of IPR versus BPM

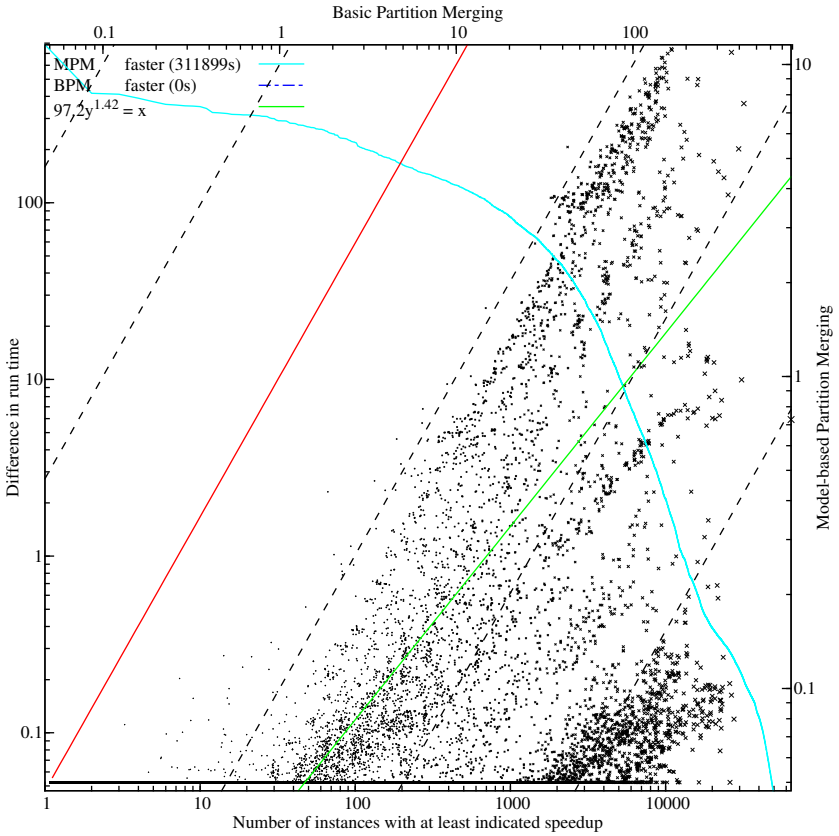


Fig. 31. Run time of MPM versus BPM

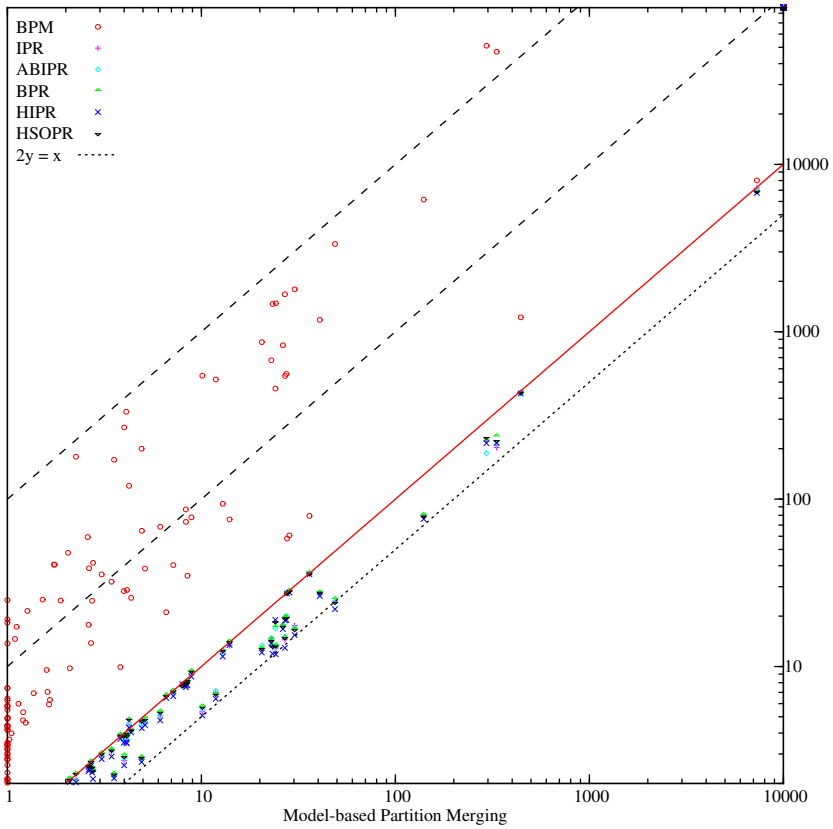


Fig. 32. Analysis time relative to MPM