Multi-Core Unit Propagation in Functional Languages

Jonathan Alexander Leaver
The University of Western Ontario

Supervisor
Dr. Robert E. Mercer
The University of Western Ontario

MULTI-CORE UNIT PROPAGATION IN FUNCTIONAL LANGUAGES

(Spine title: Multi-core Unit Propagation in Functional Languages)

(Thesis format: Monograph)

by

Jonathan Leaver

Graduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

THE UNIVERSITY OF WESTERN ONTARIO

School of Graduate and Postdoctoral Studies

CERTIFICATE OF EXAMINATION

Supervisor:	Examiners:
. . . .	. . . .
Dr. Robert E. Mercer	Dr. Graham Denham
	. . . .
	Dr. Marc Moreno Maza
	. . . .
	Dr. Stephen M. Watt

The thesis by

Jonathan Alexander Leaver

entitled:

Multi-core Unit Propagation in Functional Languages

is accepted in partial fulfillment of the

requirements for the degree of

Master of Science

. . . .	. . . .
Date	Chair of the Thesis Examination Board

Abstract

Answer Set Programming is a declarative modeling paradigm enabling specialists in diverse disciplines to describe and solve complicated problems. Growth in high performance computing is driving ever smarter and more scalable parallel answer set solvers. To improve on today's cutting-edge, researchers need to develop increasingly intelligent methods for analysis of a solver's runtime information. Reflecting on the solver's search state typically pauses its progress until the analysis is complete. This work introduces methods from the domain of parallel functional programming and immutable type theory to construct a representation of the search state that is both amenable to introspection and efficiently scalable across multiple processor cores.

Keywords: Answer set programming, functional languages, parallel programming, satisfiability theory, immutable types, multi-core solvers

Acknowlegements

Two are better than one, because they have a good reward for their labor. For if they fall, the one will lift up his fellow; but woe to him who is alone when he falls, and doesn't have another to lift him up. - Ecclesiastes

In this spirit, I would like to thank my supervisor, Dr. Bob Mercer, for his endless patience and attentiveness in the face of whichever topic excited me from one week to the next - no matter how whimsical or dry it was in actuality - something I feel has an immeasurable bearing on the character and outcome of this work. My professors at Western University, two of whom agreed to be my departmental examiners, inspired this unique research by sharing their passion for functional programming, parallel computing and artificial intelligence.

Over the course of this research, I also had occasion to work with our colleages at the University of Potsdam, Germany, whose open and kind welcome to an inquisitive newcomer in this field was deeply appreciated.

I would additionally like to acknowledge my family for their tireless support and encouragement throughout this entire undertaking. I am truly grateful, for without them it would not have been possible. And finally, I reserve a special word of appreciation for my mice: Melissa, Sophie, and Florence; and my puppy Alexander. Their unconditional love made even the hardest day a joy.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This research centers around a unique confluence among the fields of logic programming, distributed systems and programming languages. Its driving factors include the advent of ubiquitous many-core computing, languages designed for parallel processing free of side effects, and tremendous demand for the benefits of these capabilities from the logic programming community.

In this context, we examine the most costly aspect of modern solvers, their unit propagation engines. This thesis presents two concurrent models, each centered around a different locus of control. The first, based on a clause-locus, represents the most natural extension of existing implementations into concurrency. The second, based on a variable-locus, draws inspiration from our ongoing research in the area of enclosed two-watched literal inversion. The ongoing work seeks to fully enclose the structure of a logic problem using functional constructs, freeing it from the underlying data. Grounded in lambda calculus, its patterns of partial application show themselves in the immutable remainders that are the foundation for this second technique. We comparatively evaluate the merits of both models and outline their implications for future work.

1.1 Answer Set Programming

Answer Set Programming (ASP) is a type of declarative logic programming, similar in syntax to Prolog, that enables its user to describe the domain and constraints of an arbitrary problem [1]. Once a problem has been described as an input program, it can be solved by any one of a variety of solvers. The runtime procedure used to satisfy this input program is decided automatically making use of sophisticated search techniques and heuristics. Some solvers are optimized for sequential operation on a desktop computer whereas others are suitable for large scale scientific clusters. In practice, an appropriate solver is selected based on the expected difficulty (i.e. solving time) of the input program.

1.2 SAT Solving and CNF

An important field of related research is the study of the Boolean Satisfiability Problem, or SAT problem. Modern SAT and ASP solvers use many of the same techniques to explore the search space of their problems, and much of the following research is developed using SAT terminology.

Given a logic formula in conjunctive normal form (CNF), a SAT solver attempts to find an assignment for variables of the formula so that all clauses in the CNF program can be satisfied. For every assignment, if some clause of the input program cannot be satisfied, then the input program is said to be unsatisfiable. For example, we consider two clauses of the boolean CNF formula $(A \vee B) \wedge (\neg B \vee \neg C \vee D)$. For this formula, at least one of the literals A or B must be true in the first clause. Similarly, at least one of the literals $\neg B$, $\neg C$ or D must be true for the second clause.

1.2.1 DIMACS Format

To a SAT solver, input problems are most often represented in the DIMACS CNF format. Variables of the conjunctive formula are assigned integers in this representation. Where literals of the problem appear in clauses, their integer variable is presented with a positive or negative sign to carry the literal's polarity. Individual clauses are post-fixed with the unsigned value 0 in place of the delimiting conjunction operator [2].

The formula $(A \vee B) \wedge (\neg B \vee \neg C \vee D)$, above, would be represented in DIMACS as the sequence 1 2 0 −2 −3 4 0. This representation is important for two reasons. First, it illustrates the deceptive simplicity of an input SAT problem. Second, this representation is carried forward into the implementation details of nearly every modern solver's intermediate data structures.

1.3 Search

In this context, search techniques are strategies for selecting literal values to assign true or false in order to satisfy the formula under consideration. Several techniques will be introduced in the following sections, including DFS, DPLL, CDCL and Parallel CDCL.

1.3.1 Depth First Search

One of the best known algorithms in computer science, depth-first search (DFS) can be used as a simple method for obtaining these assignments. An example implementation of DFS might start by assigning variables to true, one at a time, until either some clause cannot be satisfied or a solution is found. If a clause cannot be satisfied, the specific instance of this assignment is in conflict. Under conflict, depth-first search will attempt a false assignment for the most recently assigned variable. If the conflict persists, it will clear that variable and attempt to falsify the preceding variable of the assignment. Once a conflict has been resolved, it continues forward with additional true assignments. When the first variable, or root of the search tree, results in conflicts under both true and false assignments, the problem is said to be unsatisfiable.

This assignment procedure is illustrated in Figure 1.1, which backtracks when a conflict is encountered. In this example, DFS is unable to assign the literal D at the leaves of the

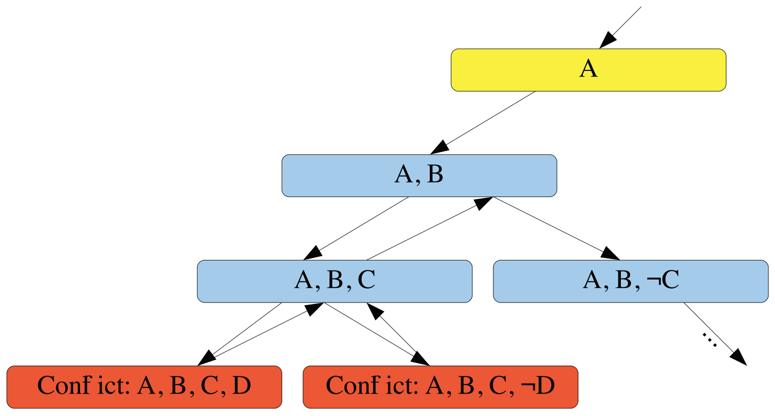

Figure 1.1: Depth First Search Assignment

tree under partial assignment $\{A, B, C\}$. Given its tree structure, search procedures based on DFS can be easily implemented recursively.

1.3.2 DPLL

Modern search algorithms are descendants of the well-known Davis-Putnam-Logemann-Loveland (DPLL) procedure [3][4]. It uses additional information encoded in the structure of the formula to integrate the consequences of an assignment decision, called unit propagation, discussed further in Section 1.3.3. Based on a depth first search, it is outlined as a recursive function in the F# programming language below:

```
let rec DPLL assignment =
    if containsAnUnsatisfiableClause() then false
    else if isCompletelyAssigned() then true
    else
        for clause in unitClauses()
            propagate(unitLiteralOf clause)
        let decision = chooseUnassigned()
        if DPLL(assignment + decision) then true
        else DPLL(assignment + not(decision))
```

It recursively assigns values to unassigned variables in the input formula. If a complete assignment can be made, the problem is solved. It will backtrack each time an assignment causes a clause of its input formula to become unsatisfiable. It then tries the opposite assignment at each level, backtracking further if neither assignment is suitable for a solution.

The notable extension to DPLL from DFS is the propagation of unit literals from unit clauses on lines five and six of the preceding algorithm. Central aspects of this procedure are outlined next.

1.3.3 Unit Literals and Propagation

Recall the CNF formula $(A \vee B) \wedge (\neg B \vee \neg C \vee D)$ discussed in Section 1.2. The propagation of unit literals at each decision is an important aspect of this procedure. A SAT formula in CNF is the conjunction of all clauses and the disjunction of all literals within each clause. For this formula to be satisfiable all clauses must be satisfied, but only one literal from each clause needs to be satisfied. This principle is the basis of unit propagation. If an assignment has eliminated all but one literal in a clause, the only remaining literal - no matter how improbable - must be true [5]. This remaining literal is called a unit, and the process of propagation adds it to the partial assignment.

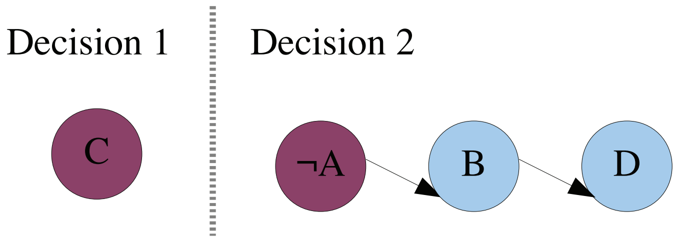

Figure 1.2: Unit Propagation From Partial Assignment

Under assignment $\{C, \neg A\}$, for instance, we might rewrite the example CNF formula as simply $(B) \wedge (\neg B \vee D)$. By process of elimination, the literal B becomes a unit literal of the first clause. It must be true under this assignment, leaving the partial assignment $\{C, \neg A, B\}$. Propagating B, consequently reduces the formula to $(B) \wedge (D)$, leaving D as a unit in the second clause. Without further search, unit propagation helps the solver move from a partial assignment of $\{C, \neg A\}$ to a satisfying assignment of $\{\neg A, B, C, D\}$ for this formula.

1.3.4 Recursive Clause Rewriting

Some early descriptions of the Davis-Putnam-Logemann-Loveland procedure are defined recursively such that reduced incremental state is passed forward during search and then restored on backtracking. Implementations of this type treat input problems as a mathematical set of clauses wherein each clause is a set of literals. Literals are removed from each clause in which they appear false. Similarly clauses with at least one true literal are removed from the problem set. This technique is used in the example from Section 1.3.3 when illustrating unit propagation.

These recursive algorithms backtrack when any clause becomes the empty set, having eliminated all possible literal assignments capable of satisfying that clause. Similarly if the problem itself becomes the empty set, then all clauses are seen to have been satisfied by the working assignment. Search would then stop and the resulting assignment would be returned.

Memory, several times the initial problem size, is required to store intermediate configurations and this is an important reason why this approach is not typically used in practice. It does, however, reduce the working size deep in search by removing from consideration all clauses and literals whose state has been decided.

This type of rewriting is not used by modern solvers which instead use the two-literal watch scheme, but its pattern is nonetheless important going forward.

1.3.5 Two-Literal Watch Lists

The use of two-literal watch lists is a technique crucial to high performance unit propagation. It dramatically reduces the computational overhead of detecting both unsatisfiable and unit clauses [6].

The procedure for maintaining watches relies on two simple assumptions. First, if a literal being watched becomes unsatisfied, any other literal of that clause that is not already unsatisfied will be watched instead. Second, each clause must be watching two different literals. Watches are placed on lists for each literal, so that a solver will visit only those clauses whose watches are affected by an assignment.

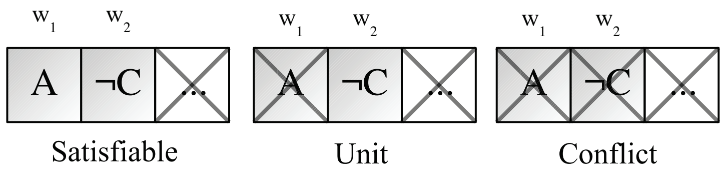

Figure 1.3: Reading Two Watches

If these conditions are maintained under this procedure, the state of a clause can be derived simply by evaluating its two watched literals. The three possible scenarios are illustrated in Figure 1.3. Having eliminated all other satisfiable or unassigned literals within the clause, if both watches are left unsatisfied, then the clause is in conflict with the current assignment. If, however, only one is left unsatisfied then the other watched literal is unit for that clause. In all other cases, the clause is still satisfiable and requires no further attention under this assignment.

1.4 Modern CDCL Solvers

Tremendous advancements have been made over the traditional DPLL procedure in the last decade leading up to today's conflict-driven clause-learning (CDCL) solvers. Most important among these breakthroughs is a technique to calculate the sequence of unique implication points (UIPs) at the moment when an assignment renders some clause unsatisfiable [7]. The first UIP of this conflicting assignment enables solvers to construct a new clause that most precisely describes its source. Consequently, a solver can use this new clause to back-jump to a much earlier point in the search and avoid future assignments leading to this conflict [8].

State-of-the-art solvers continuously generate new conflict clauses to skip over entire regions of a problem's search space. To facilitate this, they use sophisticated heuristic techniques, such as Berkmin [9] and VSIDS [6], to select literals for assignment that are most frequently involved in conflicts. This can dramatically reduce the time to compute a solution. While this shift in design de-emphasizes the rigid binary search used by DPLL, the solver now needs to actively manage a clause database of monotonically increasing size. This is a fundamental time and space trade-off underscoring the importance of efficient unit propagation and watch maintenance across the clause database.

As the space required to store clauses grows continuously, modern CDCL solvers employ various strategies to prune their learned clause database [6]. For practical reasons, a solver should retain only those learned clauses which prove useful to its ongoing search efforts. Unfortunately, calculating a clause's level of abstraction to determine how much of the search space it defers is computationally difficult. Clause length [10] and literal block distance (LBD)[11] are common techniques that have met with some empirical success. Other techniques evaluate the degree to which a clause contributes to the generation of new conflicts. Clauses that frequently participate in the implication graph leading up to a conflict and those comprised of literals with high heuristic value are more likely to be retained.

1.5 Parallel CDCL Solvers

Early parallel solvers based on DPLL focused on distributing the recursive branches of their search among parallel nodes in order to reduce the time to compute solutions [10][12][13]. This is analogous to the cliché horror movie phenomenon of splitting up to search for a missing comrade. Although this mode is still supported in modern parallel CDCL solvers like Claspar, recent work has focused on using competing solver instances with differing parametric characteristics [14]. Restart policies [15][16][17], heuristics, and clause deletion strategies [11] can all be modified to give multiple concurrent CDCL solvers different behaviors when faced with a previously unseen problem. These two major parallel search strategies are contrasted here in Figure 1.4.

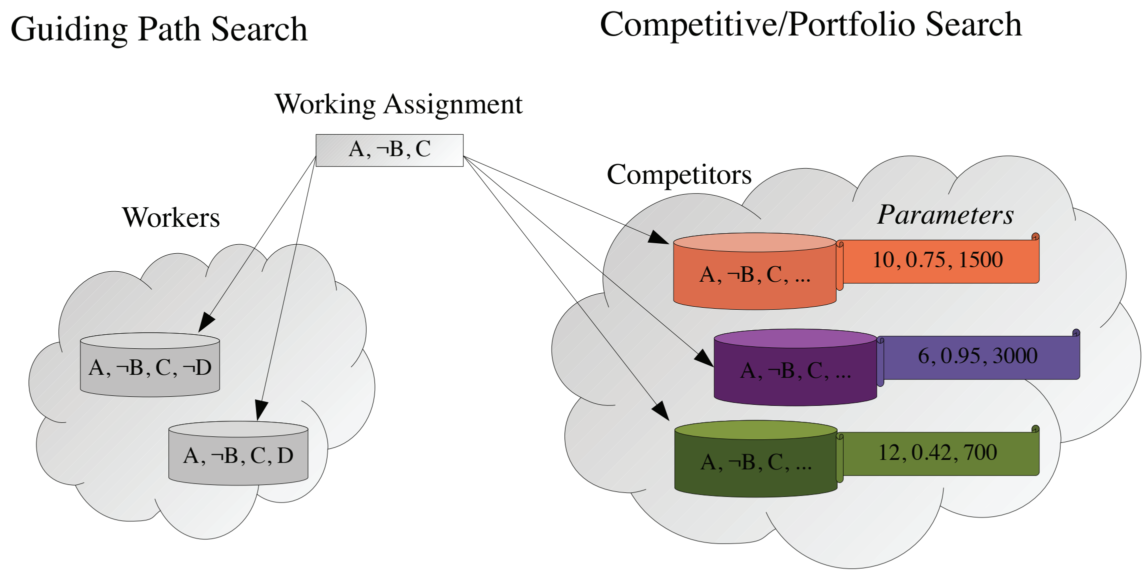

Figure 1.4: Guiding Path vs. Portfolio Search

As a consequence of this competitive portfolio mode, the different behaviors for each concurrent solver can result in different conflict clauses. Since a learned clause is a problem-scoped statement about some combination of literals, it is valid no matter where a solver is in the search space. As a result, cutting-edge CDCL solvers include various mechanisms for cooperatively exchanging and integrating clauses among competitors [10][14].

Although the potential benefits of clause exchange are clear, numerous challenges exist. Learned clause databases may largely intersect if concurrent solvers reach similar conflicts. Duplication of clauses in memory may also influence the maximum size of a clause database and consume unnecessary throughput during exchange. Moreover, the rate of discovery for new clauses may exceed the competitor's ability to successfully transmit them. Although advanced techniques for selection and flow control exist, the opportunity cost of selecting, transmitting, and integrating clauses is time a solver would have spent searching for solutions [18].

1.6 Summary

Having provided a brief look at the core topics underlying the domain of this research, a further examination of its motivation will be provided. Further aspects of systems architecture, functional languages, logic problem solving and parallel programming will be introduced as the overall strategy is developed and presented. These core topics will be expanded and additional literature introduced to support our conclusions as new elements are brought to the surface. Detailed coverage of the employed methodology will be provided along with a discussion of the measurements which were taken and the implications of those results. Finally, a discussion of the unresolved issues as potential future work will be concluded with an overview of our intended contributions.

Chapter 2

Motivation

The last decade has seen remarkable strides in all three areas surrounding this research: logic problem solving, functional languages, and multi-core programming. What follows is a brief introduction to these in order to address the impetus for this research, what changed, and why this topic becomes important today.

2.1 Systems Architectures

With the recent acquisition of forty-eight-core compute nodes at the University of Western Ontario, the challenge of coordinating competing solvers within a single process is motivating fundamental changes to the architecture of our parallel solvers. The duplication of clause databases among competitors and the problem of memory locality and contention to each of the four twelve-way processing units are key issues influencing this work. In order to operate effectively on these systems, data structures and the patterns for processing them need to be carefully studied.

2.1.1 Memory Contention

The problem of memory contention has been a topic of ongoing discussion in the community for several years. Concrete measurements were published in 2009 illustrating the magnitude of various memory inefficiencies as they apply to a variety of then-current logic problem solvers [19].

Today, this problem is magnified as processor manufacturers have multiplied the number of available cores without correspondingly dramatic expansion in memory access. Quad-socket Opteron servers, like the compute nodes of Western's new cluster, provide four channels to separate banks of system memory. As a consequence, there exists a significant contention among the CPU cores for access to uncached memory at a ratio of nearly twelve to one.

Resulting from ongoing work in this area, CDCL solvers aggressively apply heuristics to prune their clause databases with a preference for those clauses most often involved in conflicts. The multi-core version of Clasp 2.0 uses this technique in portfolio mode to maximize its performance by maintaining a small local working set for each of its concurrent workers. The intention is to retain as much local solving state in cache as possible while avoiding clauses that act as dead weight – a strategy which has proven effective in competition [20].

Multi-core Clasp 2.0 was also studied by our research group as part of a landmark paper on the patched hybrid-core of Platypus 0.2.8. These studies were carried out on twenty-four core Magny-Cours Opteron servers in varying configurations. A benchmark summary of solution time for *sequence1-ss1* through *sequence4-ss4* from the measurements taken for this paper appears here in Figure 2.1. It illustrates both that scalability in portfolio search can be achieved but that the reduction in solving time per core on complex problems can vary non-linearly with additional compute power.

2.1.2 Multi-core Runtime Analysis

The sophistication of a solver's runtime analysis and decision-making will grow beyond just heuristics for clause retention and pruning. Clause exchange, as discussed previously, is an ongoing area of work. Even more recent work on clause freezing and activation attempts to take these activities to a new level [21]. On one hand the availability of multi-

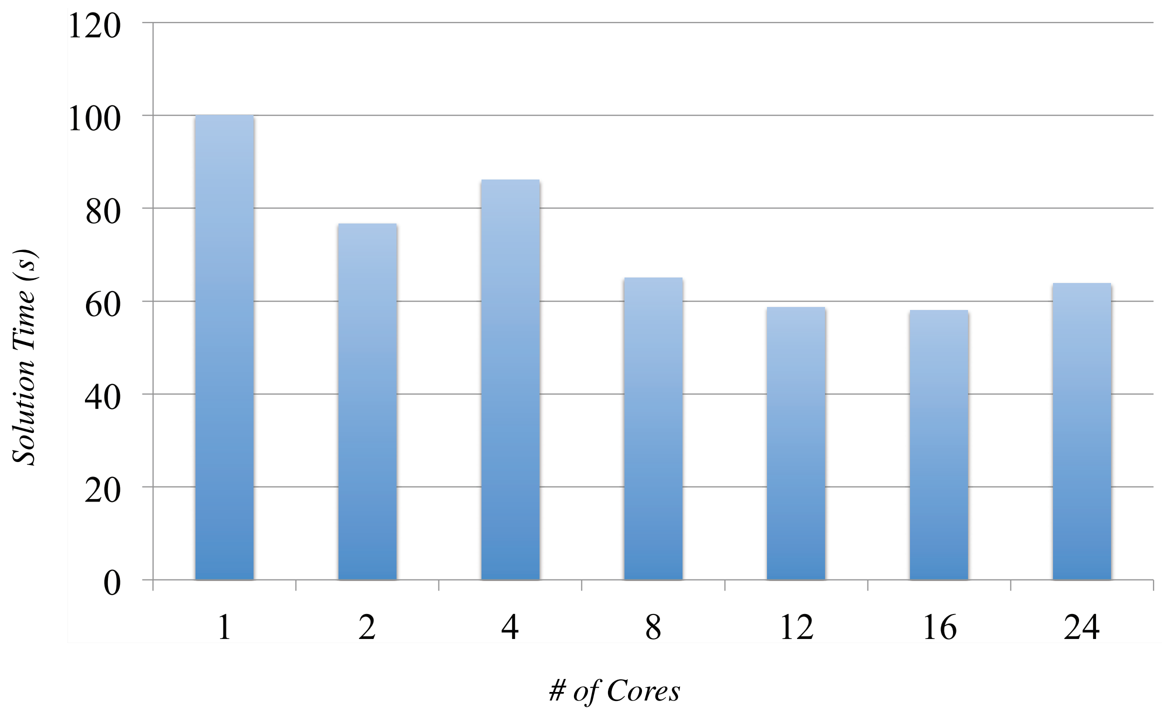

Figure 2.1: Multi-core Clasp 2.0 - Sequence Benchmarks

core chips - up to 16 cores for mainstream x86-64 chips - offers plenty of resources for these tasks. On the other, an unfortunate side-effect of existing solver designs is that the solver execution must be interrupted for a coherent view of its state. The problem of self-reflection becomes even more problematic given not just mutation but also concurrency.

2.2 Functional Languages

The selection of an ideal research platform is an important consideration for this kind of work. Nearly all cutting edge solvers are written in C-derived languages in order to maximize performance on their target platforms. To stand out in solver competitions like the SAT-Race, it is often necessary to compromise aspects of the design in favor of the best possible speed. Isolating core functions of these solvers for parallelization is problematic due to unintended side-effects induced by their optimized implementation.

With recent advancements in parallel programming languages, we felt it was important to consider whether the experimental environment should be the same as a production environment. Moreover, improvements in runtime compilers and optimizations to memory access and task delegation in these languages offer some compelling advantages to the research.

We selected F# as a language of interest for a variety of reasons. It is an ML-family functional language derived from OCaml that is designed to operate on the proven Microsoft .NET framework [22]. It is also the first fully-integrated functional language in the Microsoft ecosystem with full platform and tool support. It enables programming free of side-effects, language features for implicit asynchronous programming, immutable datatypes and a syntax that is both condensed and expressive [23]. Non-blocking compare-and-swap (CAS) data structures are also available [24]. The .NET framework uses the same high performance runtime compilers, memory managers and libraries across a variety of languages including C# [25]. These kinds of environments are not typical choices for SAT and ASP solvers. However, we feel that once the scalability and characteristics of various designs have been established, the precise implementation can be adapted to fit more conventional C-derived competition solvers.

2.2.1 Multi-threaded Programming

Although we felt that F# represented the most exciting choice for our theoretical research given its many compelling features, a diverse selection of alternative concurrency platforms are used by the community to achieve similar goals. For instance, multi-core Clasp 2.0 uses the Intel Threading Building Blocks (TBB) to achieve its high level of concurrent performance. The TBB library offers many powerful tools and structures to support this kind of parallel programming [26].

Other recent work towards developing a parallel version of MiniSAT has used Cilk-5 extensions to the C language and its work-stealing scheduler [27] [28]. Additional resources such as Cilk++ provide advanced concurrency to C++ software [29]. Newer work in this area includes hyper objects, such as reducers, that will enable efficient parallelization of competition solvers once appropriate strategies have been developed [30].

Hybrid solvers such as Platypus 0.2.8 which combine MPICH2 for clustered coordination with local multi-threading adapt their own custom portable threading libraries for parallel answer set solving. This enabled Platypus to support a wide variety of platforms after its introduction in 2005 [13].

2.3 MiniSAT Decomposition

A measurement of the solving time for a sampling of problems in serial MiniSat 2.2.0 has shown approximately 80% of processor cycles are spent during unit propagation and watch maintenance as opposed to a combined 20% for all other tasks. This observation suggests that experimental work focused on a scalable unit propagation engine will target the most costly task of the CDCL search procedure.

aloul-chnl11-13.cnf	91.03	cmu-bmc-barrel6.cnf	67.41
cmu-bmc-longmult15.cnf	76.01	een-tip-sat-texas-tp-5e.cnf	60.87
goldb-heqc-alu4mul.cnf	77.26	goldb-heqc-x1mul.cnf	87.10
hoons-vbmc-lucky7.cnf	68.78	manol-pipe-c6bidw_i.cnf	82.19
mizh-sha0-36-3.cnf	79.20	schup-l2s-abp4-1-k31.cnf	81.17
simon-s03-w08-15.cnf	79.03	velev-engi-uns-1.0-4nd.cnf	82.49

Table 2.1: MiniSAT - % Time In Propagation

Moreover, improved unit propagation is thought to be especially important for problems which require larger clause databases in order to compute solutions with completeness — as not all problems can be easily solved using a smaller working-set of clauses [31]. If the cost of maintaining the clause database can be reduced, the size of that database may also be increased — thus retaining a greater selection of potentially useful clauses.

Chapter 3

Strategy

The approach to this work is largely a comparative study that explores several parallel designs using functional language constructs in deviation from conventional parallel solvers. The desire is to demonstrate patterns appropriate to this environment and evaluate their merits.

Existing logic problem solvers written in functional languages tend to fall into two distinct qualitative categories. Solvers of the first type exhibit the language rather than the solver. Although they make good use of functional elements, their feature-set as solvers is often limited to classical Davis-Putman search. They are also typically sequential in design. Solvers of the second type implement the latest search strategies but with an almost monastic transcription of design patterns that have proven effective in procedural C-based competition solvers.

The goal of our research is to focus on the most costly aspect of modern solvers, to develop strategies for its concurrency and to do so in a manner that is expressed naturally in a functional paradigm so as to surface the many compelling advantages of these languages for multi-core problem solving.

3.1 MiniSAT Propagation Procedure

Since the most costly aspect of today's solvers is unit propagation, we recall that there are only a few fundamental building blocks for a unit propagation engine:

Variables are the underlying items of assignment, in the set $V = \{v_1, v_2, ..., v_n\}$ for some number of variables n, whose values are true, false, or not yet assigned.

Literals are signed occurrences of a variable in the set L, such that $\forall v_i \in V \bullet v_i \in L \wedge -v_i \in L$ having $|L| = 2|V|$. The value of each literal $l \in L$ is determined by the underlying variable's assignment in two scenarios: 1) $v_1 = false \implies l_1 = false \wedge -l_1 = true$, and similarly 2) $v_1 = true \implies l_1 = true \wedge -l_1 = false$.

Clauses are sets of literals $C = \{l_1, ..., l_c\}$ where $\forall l_i \in C \bullet -l_i \notin C$. Each clause requires at minimum one literal to be true for a satisfying assignment to exist.

However, two additional components are necessary for unit propagation engines which implement the two-watched literal scheme.

Watched Literals (two per clause) track variable assignments which may cause that clause to become satisfied, unsatisfied or unit.

Watch Lists (one per variable) retain an inventory of clauses whose watches depend on a particular variable's assignment.

Not all solvers retain the notion of assignment variables. For example, assigning some variable, say v_2, the value true is equivalent to assigning all occurrences of literal l_2 the value true and all occurrences of literal $-l_2$ the value false. It nonetheless remains a useful abstraction for our purposes in order to maximize watch list length.

The MiniSAT procedure draws unit assignments from a propagation queue which contains new units in the order they were obtained. After drawing a unit, MiniSAT updates its variable assignment and iterates sequentially over the clauses attached to that variable's

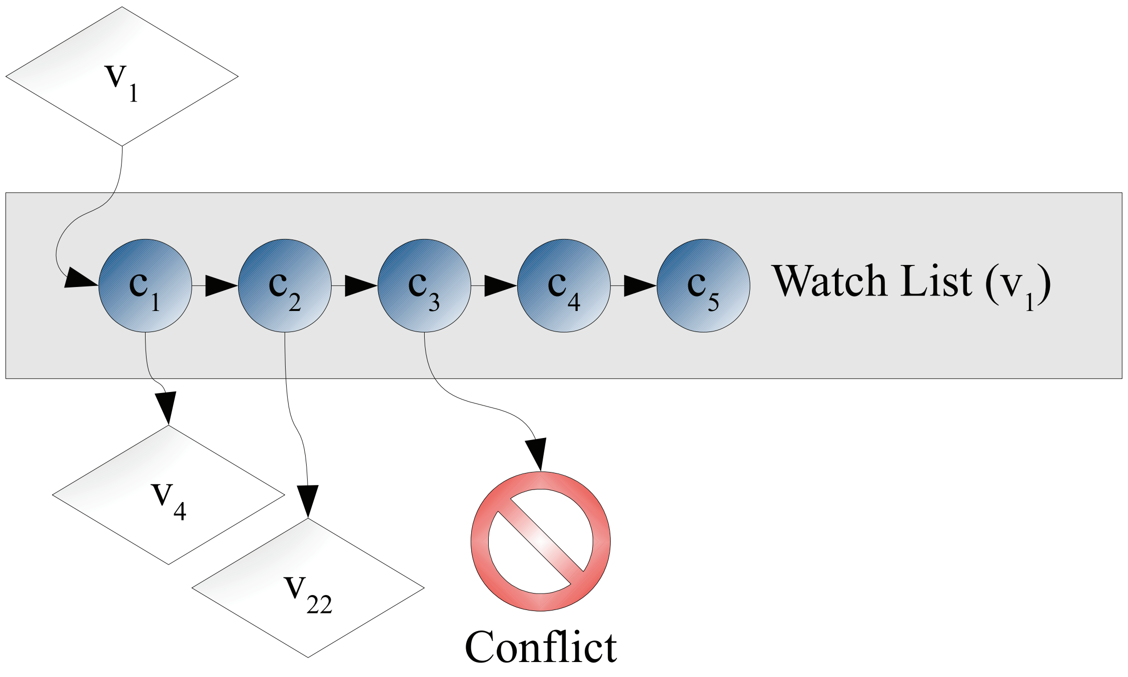

Figure 3.1: Watch List Dispatch Under MiniSAT

watch list. Each clause in turn may select new watches, add a unit to the propagation queue or raise a conflict.

In Figure 3.1, we see that when variable v_1 is assigned some value, consequences for each clause are computed sequentially. New watches are selected in clauses c_1 and c_2, but propagation aborts when a conflict is encountered in clause c_3 — thus rendering the problem unsatisfiable under the current assignment.

3.2 Locus of Control

The two-watched literal scheme presents two points of control to a concurrent propagation engine, namely the pair of watch lists to which a single clause belongs. In order to avoid locks, it is helpful to provide a single locus of control so that no two concurrent workers will compute the state of a clause under propagation simultaneously — introducing the risk of contradictory changes.

Where should the conceptual focus of asynchronous computation be in a unit propagation engine designed for concurrent execution? To that end, there are two particularly elegant possibilities: a variable locus and a clause locus. In each, one of the two elements carries out its tasks concurrently while the other is represented as data.

3.3 Clause Locus

The clause locus technique is a natural extension of the existing mutable design employed by CDCL solvers such as MiniSAT. Its simplistic approach to parallelism is analogous to switching the iteration's *foreach* statement with *Parallel.ForEach*, given that adequate concurrency control measures are in place so that target lists are modified safely.

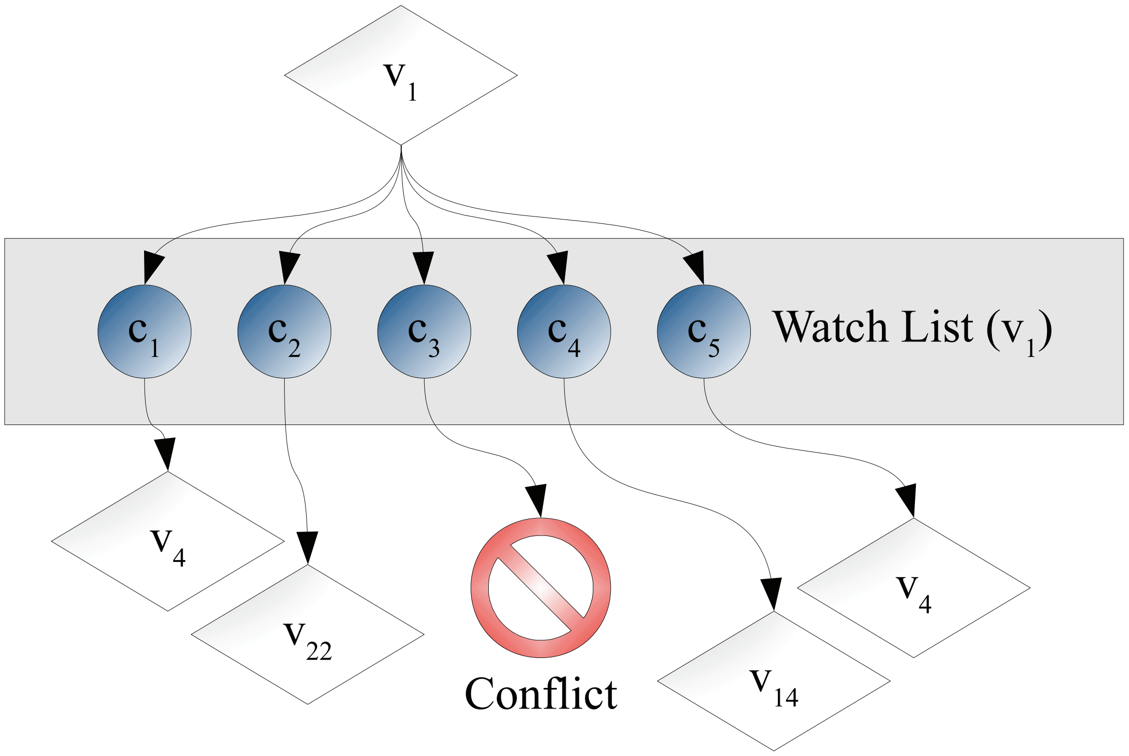

Figure 3.2: Watch List Dispatch With Clause Concurrency

As illustrated in Figure 3.2, the solver-core draws variable v_1 for assignment from the unit propagation queue. All clauses on the watch list are then invited to run their selection algorithms simultaneously. New units obtained by any clause are enqueued for subsequent propagation. In this example, however, a conflict is discovered in clause c_3 that would terminate unit propagation and begin conflict analysis. Notice also that both clauses c_1 and c_5 have selected v_4 as their next watch to replace v_1. As a consequence, care must be taken to guarantee that techniques such as locking or the use of special non-blocking (CAS) data structures prevent race conditions altering this list.

It is clear from this example that the degree of concurrency is also limited in part by the length of the watch list for v_1. For instance, a sixteen core machine may have limited work during some propagation steps. In both designs, watch lists are held by assignment variables instead of literals for this reason. So compared with literal-based designs, we have simply:

$$List_{v_1} = List_{l_1} + List_{-l_1} \implies |List_{v_1}| = |List_{l_1}| + |List_{-l_1}|$$

The clause-centric approach conceptually partitions watch lists for concurrent operation. It would seem reasonable to expect that better scalability will be achieved on problems with a higher clause to variable ratio. Moreover, it is worthwhile questioning whether this approach is natural given typical functional constructs. For instance, the representation of lists will vary. Although this design extends quite naturally from traditional implementations, it may or may not be suited for this environment.

3.4 Variable Locus

The proposed variable locus derives from a desire to optimize the unit propagation engine for immutable types and implicit threading, having studied the strengths and limitations of the preceding clause locus strategy. In this model, each variable is conceptually independent — potentially carrying out tasks concurrent of the others. Communication between variables is carried out with the assumption of asynchrony, and can be viewed as messaging between them. Immutable data types, as they appear in F#, will be further discussed in Section 4.2; however, the theoretical structures are introduced here.

3.4.1 Conflict Detection

A clause of length c will be read from input in DIMACS format as a string of literals, formally $[l_1, l_2, ..., l_c]$, for some selection of literals up to the length of that clause. Using an immutable list data-type, this clause can be represented as a sequence of *cons* cells in the form $\langle l_1, \langle l_2, \langle ..., \langle l_c, \epsilon\rangle\rangle\rangle\rangle$. In memory, the list will be stored recursively as a literal element with a pointer to the next *cons* element in its sequence. Here ϵ is used to represent the empty list — that effectively no further elements exist in the sequence. For each *cons* cell, we call the first element the *head* of the list, and the second element its *tail*.

Let each clause be identified by a unique integer identifier x corresponding with its sequence in the input file. In the above representation, we use the immutable tuple (or ordered pair) data-type to carry the unique identifier for a clause and its immutable list as $(x, \langle l_1, \langle l_2, \langle ..., \langle l_c, \epsilon\rangle\rangle\rangle\rangle)$. In this form, we will call the immutable list for a clause its *remainder*, i.e. for (x, y) where y is an immutable list, y is the remainder.

Suppose further that each clause has exactly one watched literal rather than two — a supposition to be retracted shortly. Given the recursive nature of the immutable list, it is simplest to let this watch literal be l_1.

The variable locus assumes that variables during unit propagation will be able to operate independently and simultaneously of each other. If we let v_1 be the assignment variable for l_1 and $-l_1$, the single watch assumption allows for a guarantee that any clauses with watches on l_1 or $-l_1$ can be held exclusively on the watch list for v_1. Furthermore, the *remainder* of every clause on the watch list for v_1 contains either l_1 or $-l_1$ in its *head* (or first) position.

When assigning a value of true or false to v_1, a reason for the assignment is recorded for conflict analysis. It is either an assumption or a unit implication from some other clause. In either case, the following rules apply regarding the watch list:

1. Any clause whose watch literal is true under this variable assignment is dropped.

These clauses are satisfied under this assignment and require no further consideration along this line of reasoning.

2. All others are reduced exactly one recursive step. For the *remainder* of this clause, we obtain its *tail* and construct a new tuple with the tail replacing the former remainder, such as $(x, \langle l_2, \langle ..., \langle l_c, \epsilon \rangle\rangle\rangle)$. For all clauses thus reduced, a new watch must be selected. There are precisely two possible patterns for the *tail* of this tuple:

   (x, ϵ) The remainder is empty. This clause has passed through variables for every literal in its list and found each literal to be false. As a consequence, there are no possible assignments along this line of reasoning to any literal in this clause capable of rendering it satisfiable. A conflict has been obtained and unit propagation will stop. Control returns to the solver-core to analyze the conflict on clause x, to invalidate any assumptions and to resume search for satisfying solutions if possible.

   $(x, \langle l_2, ... \rangle)$ Let the new watch literal be l_2. Assignment variable v_1 asynchronously sends this new tuple to v_2. If v_2 has already taken a value, then this watch list procedure is recursively applied to $(x, \langle l_2, ... \rangle)$ on receipt at v_2.

The path of clause x is illustrated in Figure 3.3. Introduced in the first step, it is exchanged between v_1 and v_2 then v_2 and v_3, as each literal is falsified by assignments to these variables. It then becomes a source of conflict in the final step as l_3 is falsified by assignment to v_3.

The procedure as outlined above is sufficiently powerful as to detect any conflicts during search, but is inadequate to detect new units. For example, let us introduce the clause $(x, \langle l_1, \langle l_2, \langle l_3, \epsilon \rangle\rangle\rangle)$. If we falsify v_1, x is delivered to v_2. If subsequent reasoning falsifies v_3, then the only possible assignment for x is that l_2 must be true. However, it is not efficiently possible under this procedure to detect that the state of clause x on a watch list for v_2 has changed — since v_2 remains unassigned at this time.

The solution applied by sequential propagation engines and which was also safe in

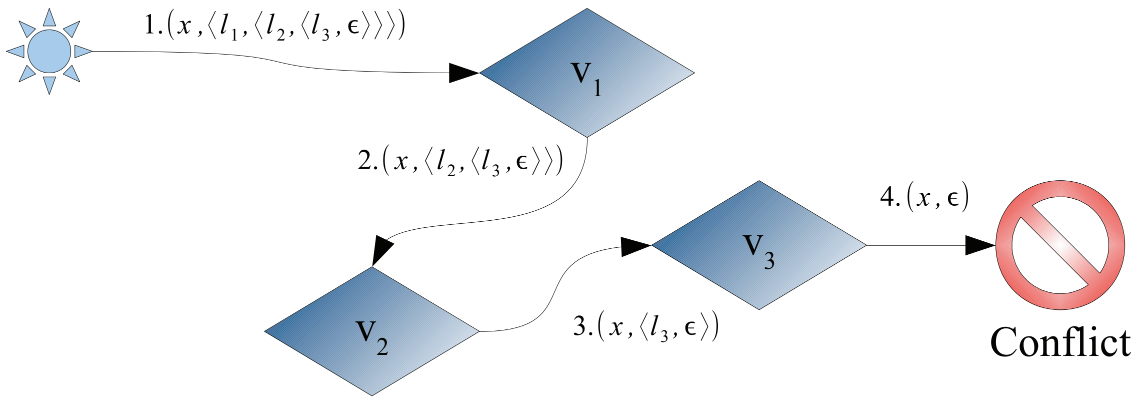
1.$(x,\langle l_1,\langle l_2,\langle l_3,\epsilon\rangle\rangle\rangle)$
V_1
2.$(x,\langle l_2,\langle l_3,\epsilon\rangle\rangle)$
4.(x,ϵ)
V_3
V_2
3.$(x,\langle l_3,\epsilon\rangle)$
Conflict

Figure 3.3: Single-Watch Conflict Under Variable Concurrency

the clause-oriented design does not work here as defined. Normally a second watch on a clause is introduced to detect when all other literals have been eliminated. In the variable locus, however, this could introduce two points of control which must continuously be synchronized for each active clause in the program.

3.4.2 Unit Detection

If the complete state of clause x is evaluated at v_2 under concurrent execution, it is clear that all literals preceding l_2 have been eliminated. Unclear, however, is whether the state of literals subsequent to l_2 has been decided. Any attempt to evaluate this state from v_2 will be nondeterministic in its result. The value of l_3 may be held constant, or may be in a state of change. A method to safely determine this residual state is needed.

Recall that clauses are introduced to the system in the form $(x,\langle l_1,\langle l_2,\langle ...,\langle l_c,\epsilon\rangle\rangle\rangle\rangle)$. Let us now introduce a recursive inverse for this same clause so that it traverses variable assignments in the opposite direction. We simply reverse the *cons* list so that clause x takes the form $(x,\langle l_c,\langle l_{c-1},\langle ...,\langle l_1,\epsilon\rangle\rangle\rangle\rangle)$ instead. Here, literal l_c becomes the watch literal for clause x and, when falsified, issues the remainder to v_{c-1}.

From our previous example, a clause in this form traverses precisely the sequence of variables whose state had been nondeterministic, arriving at v_2 having eliminated those literals as valid assignments.

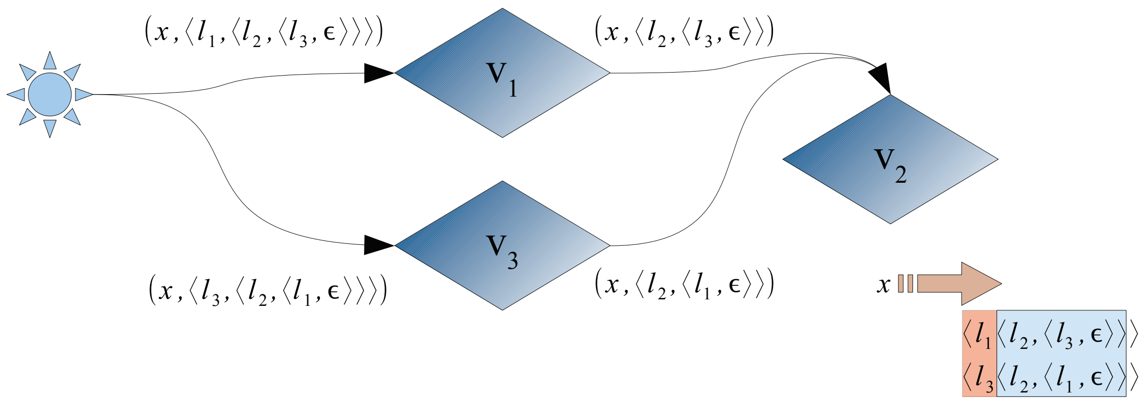
$(x,\langle l_1,\langle l_2,\langle l_3,\epsilon\rangle\rangle\rangle)$
$(x,\langle l_2,\langle l_3,\epsilon\rangle\rangle)$
V1
V2
V3
$(x,\langle l_3,\langle l_2,\langle l_1,\epsilon\rangle\rangle\rangle)$
$(x,\langle l_2,\langle l_1,\epsilon\rangle\rangle)$
x
$\langle l_1\langle l_2,\langle l_3,\epsilon\rangle\rangle\rangle$
$\langle l_3\langle l_2,\langle l_1,\epsilon\rangle\rangle\rangle$

Figure 3.4: Two-Watch Unit Under Variable Concurrency

The solution to the requisite two-watch literals pattern, then, is to introduce both clauses to the system. To detect if clause x is unit at v_2, it need only know that both instances of the clause have been received. This information safely eliminates the possibility that any other literal might satisfy this clause under the current line of reasoning. Consequently, v_2 can immediately take the value which renders l_2 true for this clause. Clause x will then be recorded as the reason for its assignment — to be used for calculating resolvents during any subsequent conflict analysis.

Illustrated in Figure 3.4, it becomes clear that by traversing the clause from both ends, v_2 is able to eliminate all preceding and succeeding literals to l_2.

3.4.3 Speculative Unit Propagation

An exciting advantage to any parallel design of this type is the possibility of speculative unit propagation. During the propagation of each unit literal across the clause database, additional unit literals can be discovered that are within scope of the current assignment. In this event, it will be possible to concurrently delegate these for continuous propagation without directly waiting for further instructions from the solver core. This reduces idle time, better balances work load and enables the solver to encounter conflicts faster. Moreover, a variable-centric design will detect and raise conflicting unit assignments as they occur since the point of control for a variable's assignment is safe under concurrency.

3.4.4 Watched Clause Rewriting

In particular, memory usage is an important aspect of this design. Traditional watch management alters the state of the clause database for all subsequent search without regard to backtracking. That is, if a watch is moved just prior to a conflict, its change is retained even after back-jumping has completed. Conventional wisdom regards this as positive, since watches are cleared along the most-worn path to conflicted areas of the search space. This fits well with modern heuristics like VSIDS which develop a preference for traversing highly conflicted literals [6]. The clear opportunity cost, however, is that watches have been relocated to other literals, which will be encountered as the solver broadens its search. Moreover, the rapid and flexible changes exhibited by heuristics like Berkmin may further alter this dynamic [9].

Our proposed design revives aspects of Recursive Clause Rewriting, introduced in Section 1.3.4. In particular, the use of immutable data structures offers a compelling, low-cost restoration of both clauses and watch state on back-jumps. The incremental state accumulated forward of the jump target can be dropped in favor of the target's prior configuration.

This strategy has several potential advantages in its own right. Watch dispersion becomes a function of the ordering for literals in the underlying problem. Dispersion can remain relatively consistent, even in an environment of rapidly changing assumptions. The use of immutable types also enables a continuous reduction in the number and size of clauses as the set of assumptions and consequences grows deeper in search. For systems with narrow access to memory, such as the Opteron servers discussed previously, the forward reduction in problem size and consistency with regard to the underlying immutable structure may improve the use of local cache.

Although it might seem that generating watched clause reductions could be costly, immutable types are used whose intermediate state is nothing more than references to intermediate *cons* cells of the initial clause. Injection of new clauses is similarly trivial: from the point of insertion, variables exchange the clauses until the watch configuration settles into place.

3.4.5 Advanced Search Strategies

Another potential advantage of this immutable design is the ease with which it can be extended to parallel portfolio and guiding path search. Once the initial program structure has been loaded, the clauses and variables it defines can be shared directly between workers without additional memory allocation. Subsequent work by competing solvers is retained in the local state for each competitor. As a consequence, the initial memory allocation is small, i.e. exactly one instance of the problem. We compare this with the size needed by mutable solvers, which is linear in the number of competitors. This is because each mutable solver continuously alters its underlying clause database according to its state of search.

3.4.6 Considerations

At issue is whether this strategy improves memory access, locality and cache behavior. Moreover, do its benefits outweigh its costs? Some of these aspects can be evaluated in terms of overall performance whereas others need more focused measurement.

The degree to which this design is sensitive to the number of variables in the underlying problem is also a consideration. As compared with the clause locus propagation engine, we suspect a relationship in the ratio of variables to clauses that may influence the effectiveness of either strategy.

Chapter 4

Method

In order to ascertain the strengths and weaknesses of the outlined strategies, several aspects of the methodology need to be established. In particular, it will be necessary to test the unit propagation algorithms within their underlying functional environment. It is common practice in the answer set and logic programming community to use benchmarking as a technique to measure the effectiveness of solvers and engines. In the following sections, we outline the benchmarking approach adopted for this research. We discuss aspects of the unit propagation engines to be benchmarked, including some details of the F# functional language in which they are written. Selected systemic and search measurements will be reviewed. Moreover, parameters, both those which are varied and those which are specifically held constant, will be given special consideration. Finally, the tools used for analysis of the results will be introduced.

4.1 Benchmarking

Benchmarking will involve running several iterations of a reference problem that is publicly available under a variety of configurations. Since the focus of research is on unit propagation, a full answer set solving engine is unnecessary. Consequently, the unit propagation engines will be benchmarked with DIMACS format CNF satisfiability problems rather than answer set formats from grounders Lparse or Gringo. In keeping with benchmarks obtained from Platypus, Claspar and Clasp for a related study, a minimum of five iterations will be taken per configuration in order to produce results of comparable quality. The benchmark configurations that follow culminated in thirteen-thousand measured runs taken over a six month period.

4.1.1 Selections

Suitable DIMACS formatted benchmarks are available from a variety of sources. In addition to those held in a repository with our colleagues in Potsdam, Germany, SAT competitions publish their benchmarks to the community in order to both maintain transparency through reproducible measurement but also to help improve the state-of-the-art in logic problem solvers. In order to maximize the utility of our measurements, we elected to use problems obtained from SAT competitions in 2002 and 2009 [32].

Detailed profiles of the selected benchmarks appear in Appendix A, including Table A.1. Selections were drawn from all three major categories, including application, crafted and random problems. Generally, the initial problem size among the selected benchmarks is gradually increasing in terms of both clauses and variables. Outliers were chosen in order to provide larger initial variable-to-clause and clause-to-variable ratios. The ratio of initial variables to clauses, which we suspect is an important factor in any comparison of the variable or clause-centric propagation designs, is illustrated in Figure A.1.

Although the selected benchmarks may seem biased along the diagonal, established researchers in the field published findings in 1992 regarding the hardness of SAT problems in terms of over and under specification [33]. They discovered that SAT problems tend to fall into easy-hard-easy distributions such that either over or under specification of the boolean constraints imposed by clauses on their variables produce problems which are easily solved. In specific terms, too many variables and too few clauses constituted under-constrained problems for which variable assignment is comparatively easy. In contrast, over-constraining a problem with extensive and intricate clauses will produce a problem that can be equally easy, but often unsatisfiable. Challenging problems tend to fall in the middle, and the selected benchmarks cover a range within that space.

4.1.2 Platform

The proposed designs call for measurements of scalability and performance on the F# runtime environment. Implementations of the F# runtime exist on both its native Microsoft Windows, as the .NET Framework, and Unix, as mono. Unfortunately, 64-bit mono 2.10.2 as installed on Western's forty-eight-way Rocks 5.4 cluster exhibited reliability problems and was unsuitable for the sheer number of measurements to be taken. These failures occurred in mono's memory subsystem, primarily its memory allocator, regardless of which benchmark or engine was being run. In particular, highly concurrent execution with small units of work as required by the clause-centric design was the most problematic. The immutable variable-centric design, however, did produce some very compelling measurements which will be discussed in the final results.

For the majority of measurements, a custom multi-CPU system was purchased and assembled in order to demonstrate F# on the platform for which it is designed and optimized. This machine runs twin eight-core AMD Opteron Mangy-Cours 6128 processors operating at a peak frequency of 2.0GHz. These sixteen processor cores make use of 128KB Level-1, 512KB Level-2 and 12MB Level-3 cache to improve access to memory. Each core has its own Level-1 and Level-2 cache, whereas each processor has a shared Level-3 cache. On the Asus KGPE-D16 motherboard, these processors access four distinct channels to main memory. The 32GB of installed memory consists of eight 4GB server-grade Kingston registered ECC DDR3 1333 memory modules. Data between runs is backed by 2TB of primary storage. To obtain reproducible results, advanced power saving features were disabled and the clock speed of all sixteen cores was locked to the factory-rated 2.0GHz.

The 64-bit version of Windows 7 Ultimate Edition is installed with version 4.0.30319 of the .NET Framework. This edition of Windows supports two CPUs with up to 32 cores using today's hardware. All services deemed unnecessary for benchmarking were disabled, including background update and security software. A Windows PowerShell script was written to automate sequential execution of the benchmark runs and collect results over the extended measurement period with several minor revisions over its lifetime.

This highly customized platform remains unchanged during the benchmarking in order to guarantee consistent results. Having examined the operating platform, we will now review salient aspects of the F# language employed in this work.

4.2 F# Language

As previous discussed, the F# language is a modern descendant of the ML family of languages, most closely related to Objective Caml — also called OCaml. OCaml added object-oriented capabilities in 1996 to the Caml language which debuted a decade earlier. Both are dialects of ML maintained at the *Institut national de recherche en informatique et en automatique* or INRIA [34]. Developed at Microsoft Research starting in 2002, F# inherits many important features from these languages. As Don Syme describes it, ”F# is a scalable, succinct, type-safe, type-inferred, efficiently executing functional/imperative/object-oriented programming language.” Moreover, it is ”both a parallel and a reactive language.” [22] It also shares certain features such as sequence expressions and workflows with *Haskell* which form the basis of its asynchronous processing. He goes on to say that ”its approach to type inference, object-oriented programming and dynamic language techniques is substantially different from all other mainstream functional languages.” [35]

Many of the crucial language features used in the following work will be briefly introduced in order to provide useful background.

4.2.1 Immutable Types

One of the most important aspects of F# as it pertains to this research is its support for immutable data types. In particular, immutable lists and tuples are used extensively by the variable-centric unit propagation procedure.

The most common immutable type in modern languages, such as Java and C#, is the *string*. Its contents cannot be directly altered, however, operations on it create new instances which carry the consequences of those operations. In F#, this pattern is extended to both common collections and end-user classes. Although mutable versions are still available and supported, immutable types have several important advantages. Most useful among these is the coherency of concurrent reads. That is, if two threads have references to an immutable structure, changes derived from that structure result in new instances held locally to those threads, rather than to any shared state between them. This added safety eases development of certain types of asynchronous procedures and will be demonstrated in our implementation of the variable-locus design.

The theoretical model for immutable lists used in our discussion of the variable locus can be easily translated into F# syntax. Consider the remainder $\langle l_1, \langle l_2, \langle ..., \langle l_c, \epsilon \rangle\rangle\rangle\rangle$. F# replaces ϵ with the empty list []. *Cons* operations simply apply the :: operator to join an element to the front of an existing list — possibly an empty one. For example, the syntax l_c :: [] replaces $\langle l_c, \epsilon \rangle$ in the previous example. This procedure can be repeated until the remainder is fully constructed, for example l_1 :: $[l_2; ...; l_c]$. In its final form, the language would represent the remainder as $[l_1; l_2; ...; l_c]$, however, the underlying linked-list structure of *cons* cells remains [35].

The language syntax and representation of the Tuple data-type is exactly the same as presented in the variable-centric design. The intermediate clause $(77, \langle l_1, \langle l_2, \langle ..., \langle l_c, \epsilon \rangle\rangle\rangle\rangle)$ can thus be represented as simply $(77, [l_1; l_2; ...; l_c])$ in F#.

4.2.2 Map and Reduce

Operations on immutable data, then, are as simple as applying a function to some reference and obtaining the result. A useful example could be the mapping and reduction of clauses on a watch list as part of unit propagation. For example, the following syntax takes watches from a watch list and *maps* them into a collection of results.

$$\textbf{let } results = (List.map\ visitWatch\ watches)$$

A function *visitWatch* defines how a single watch would be processed. *List.map* simply applies the *visitWatch* function to each element in a list of *watches* to produce a new list of results. Once all watches have been visited, the collection of results is returned.

If, hypothetically, the results were a count of clauses to be moved - either 1 or 0 for a single watch - then a simple sum of all movable clauses might be desirable. Similar to *map*, *reduce* can apply the addition function to merge all elements as follows:

let *total* = (*List.reduce* (+) *counts*)

Reduce accepts any binary function which takes two input elements of some type and returns a single result. It then applies this function repeatedly to the list in order to reduce it down to a single value.

4.2.3 Recursion and Affinity

As may have become apparent, the usual techniques for iteration of data are largely unnecessary in F#, although they do exist. Where iteration is desired as part of a program's flow-control, however, F# prefers tail recursive functions to imperative iteration. A function that is recursive includes the keyword **rec**.

In F# it is possible to require that the program use only certain CPU cores. A bit map of the desired core assignment is provided to the framework with 1 enabling use of a core and 0 disabling it. To generate this affinity setting for some number of adjacent *cores*, consider the following function:

```
let rec generateAffinity (cores : int) =
        if cores <= 1 then 1
        else 1 + 2 * generateAffinity (cores - 1)
```

The structure of this tail recursive function can be clearly altered by the compiler to use traditional iteration on the executing machine rather than stack recursion. The result of this function is used to assign a specific sequential core affinity to the currently executing process. Requesting only the cores of the first CPU is simple:

```
let affinity = (generateAffinity 8)
let current = (Process.GetCurrentProcess())
current.ProcessorAffinity <- nativeint affinity
```

In the DPLL and CDCL algorithms, there is a tendency to write tree-recursive procedures which explore one possibility and then its alternative. These cannot be optimized by the runtime [35]. In our experience, deep search of this type will result in stack overflows. An immutable stack can be used explicitly with tail recursion to store alternatives for subsequent processing which can be easily passed into the next cycle.

4.2.4 Closures and λs

As with other functional languages, F# supports closures, partial application, and lambda functions. In particular, these are used to construct and return new functions with different values bound to their enclosing environment. Although F# also supports structured classes and values, these are a clean, simple method for structured computation. Although their usage in the experimental implementations is sparse, an application of these structures will be expanded in the discussion of possible future work, especially Section 6.2 on enclosed two-watch literal inversion.

4.2.5 Async and MailboxProcessors

When executing map and reduce, the program sequentially visited each watch and produced a collection of results. These results were then reduced to a single value. Using the *async* keyword defines special asynchronous workflows in the F# language. Effectively, it is used to define blocks of work that can be computed later. Suppose we wrap the previous *visitWatch* function with *async*:

let *visitWatchAsync* (*watch*) = *async* { *return* (*visitWatch watch*) }

In this example, *visitWatchAsync* returns a task which will visit the watch enclosed within when executed. Mapping the list of watches to asynchronous tasks builds a body of work that can be scheduled later:

$$\textbf{let } asyncWatchTasks = (List.map\ visitWatchAsync\ watches)$$

This collection of tasks enables the environment to run all watch functions simultaneously when asked to do so. The following example executes all watch tasks concurrently, waiting until they are completed to obtain the results:

$$\textbf{let } results = (Async.RunSynchronously\ (Async.Parallel\ asyncWatchTasks))$$

This demonstrates the relative ease with which data can be processed concurrently in F#.

Another important language feature, that of agents, is used in the variable-centric design for concurrent processing on variables. Conceptually, the agent is implemented as a worker communicating asynchronously with the outside world through the use of mailboxes. This is a familiar pattern in high performance computing, including such popular platforms as the message passing interface, or MPI, used by solvers such as Platypus and Claspar [13][14]. Agents are implemented with MailboxProcessors — a higher level feature detailed further in Chapter 13 of Don Syme's Expert F# 2.0 [35].

4.3 Unit Propagation Experiments

As demonstrated, the F# language offers a variety of powerful and expressive options for concise concurrent programming which make it a compelling choice for developing and testing the proposed unit propagation designs.

4.3.1 Development Plan

Experimentation with the propagation control strategies is implemented in three stages. After studying the idiosyncrasies of MiniSAT, Platypus and Claspar, our first phase of experimentation seeks to gain valuable experience developing conflict-driven, clause-learning solvers in the Microsoft .NET environment — all while staying close to familiar C-style syntax. This fully functional C# CDCL solver, derived in part from MiniSAT, provides a useful baseline for any subsequent F# implementations by demonstrating realistic estimates of potential performance on top of the .NET framework's virtual machine.

Expanding the scope of this work, the second phase implements the first of two F# unit propagation engines. Focusing on the clause-locus, this multi-core implementation ports and augments the established algorithms for watch management with safe concurrency. As the first F# implementation, performance studies for mutability and threading gather additional information about the comparative performance of these components.

The third phase of development implements the second F# unit propagation engine around the variable-locus. Strengthened by studies from the previous work, it makes conscientious use of immutable data-types and asynchronous workflows. To examine the potential for search-state introspection with immutable types, it incorporates guiding path splitting and portfolio search strategies with minimal alteration to the underlying engine.

4.3.2 Experimental Versions

Benchmarks, solutions, projects and source code for all three solvers as well as some additional resources from the discussion of future work in Chapter 6 are available with Source Forge online. Source Forge is a publicly accessible web-based repository for open-source and research software. Complete version history for all files and folders is maintained in a Subversion repository with an extensive log of the changes made during the course of this work. [1]

The majority of benchmarks can be carried out on trunk revision 365, with the last substantial commit revisions to the various projects at earlier sequence numbers. For validation purposes, check-out of this specific revision will be sufficient; however, researchers intent on further experimentation will prefer to check-out the head revision of the repository.

[1]https://mouse-solver.svn.sourceforge.net/svnroot/mouse-solver/

4.3.3 C#: Mus Musculus

Named after the *common house mouse*, a more traditional serial solver for the Common Language Runtime provides a sanity-check for performance measurements taken from the F# engines. Although a unique solver in its own right, its algorithms have been adapted specifically from the MiniSAT 2.2.0 solver and redesigned to conform with the abstractions of the .NET Framework. Measurements from this solver serve to illustrate the approximate performance one can expect running on the framework's virtual machine.

As an object-oriented solver, diagrams in Unified Modeling Language, or UML, are included in Appendix C to illustrate the general structure of this CDCL solver. These diagrams are used in conjunction with the code-generation capabilities of Pragsoft UMLStudio 7.2 to dynamically link the C# source files with their UML design models. Specifically, Figures C.1 and C.2 outline two event-driven data structures and a collection of extensions to Language Integrated Query, or LINQ, for adapting value and set semantics into the conflict-driven solver. In both cases, the underlying data-types are native to the framework with extensions focusing on the event-driven behavior needed for watch maintenance and unit propagation. The solver itself, outlined in Figure C.3, is dramatically simplified in part due to these additional structures.

The command line pattern for executing this solver is very simple. It accepts a DIMACS formatted CNF file as its sole argument and uses reasonable defaults for all other options:

$$Musculus\backslash Mouse\backslash bin\backslash Release > Mouse.exe\ [problem.cnf]$$

4.3.4 F# Clause Locus: Murinae

The first F# implementation is a natural extension of conventional methods for watch management and propagation. It is a dynamically configurable solver named after the family of *Old World rats and mice*. Incorporating the first of two propagation designs, it tests additional aspects related to the F# language and runtime in order to ensure the best possible representation of results and to provide guidance for subsequent work. It explores several facets of data-types and concurrency, including measurement of mutable and immutable implementations of conventional propagation algorithms. Comparison of the implicit concurrency features of the F# language with an explicit implementation is also included to better understand and maximize the performance of asynchronous tasks. Mechanisms for configurable processor core assignment, variable word size and logic program repetition were further included to study scalability and memory contention.

It can be executed from the command-line by specifying an input problem in DIMACS format along with a number of additional options to be discussed further:

Murinae [*problem.cnf*] [*cores*] [*repetitions*] [*implicit*|*explicit*] [*mutable*|*immutable*]

Affinity

Using a technique similar to the procedure outlined in Section 4.2.3, the processor core affinity can be adjusted parametrically to obtain measurements in varying configurations of enabled or disabled CPU cores. From the command-line, the *cores* option dynamically sets this value when the solver is started.

Repetitions

As discussed in the overall strategy for the clause-locus propagation engine, it was felt that the performance of the overall propagation design would depend in part on the length of the watch lists for each variable. As clauses are learned over time, the average watch list length will grow. For purposes of determining whether improvements in performance are due to the extra information provided by these learned clauses or the extended average length of the watch lists, a simple technique was devised to gauge the overall improvement in throughput without adding additional information about the search space.

To this end, a feature was implemented for repeating the clauses of the input program in such a way that the clause database size could be increased without providing new knowledge. In the Murinae solver, this can be controlled through the *repetitions* parameter. Consequently, the average watch list length is increased for measurement without altering the overall flow of search.

For instance, any clause that would normally discover a unit or conflict under a particular assignment during the propagation stage will be accompanied by other such clauses on the watch list being processed. As a result, the unit or conflict will still be discovered even though the number of clauses on the watch list has grown. For all other situations, the number of clauses to be processed on that watch list is increased without additional conflicts or units being generated. Although this will not positively improve the search time, it may potentially increase the measured throughput in terms of clauses propagated per second to demonstrate the effect of watch-list length on the performance of this design.

Thread Management

Since F# is still a comparatively new language to the .NET framework, it remains unclear whether the explicit allocation of non-blocking threads attached to functional work queues or the use of F#'s native asynchronous constructs will lead to better performance. Notably, the native asynchronous constructs use the framework's work-stealing thread pool and can allocate additional threads in the event of a block [24]. Although every effort is made to avoid unnecessary blocks and allocations, it may or may not be better for memory locality to use pre-allocated threads attached to lambda queues. The solver can be switched between these two modes of execution at runtime by specifying either *implicit* or *explicit* threading on the command-line.

Data Structures

Another area of tension in the design is the trade-off between mutable and immutable data structures. Since F# supports both non-blocking concurrent data structures and pure immutable data types, it is unknown how use of the immutable types will scale. In preliminary measurements, procedures written for immutability yield nearly equivalent performance, but the scalability may be better. This dimension will be important to measure for any future work involving these languages. Selecting between these techniques is simply a matter of specifying the command-line option for *immutable* or *mutable* data types.

4.3.5 Concurrent DIMACS Processor

Murinae also introduces a concurrent input file processor to translate DIMACS-formatted clauses into internal data structures used for search. This processor uses Parallel Language Integrated Query, or PLINQ, to translate between the two forms. This technique makes use of partitioning methods and declarative programming to reduce the end-to-end processing time for input files and improve the overall iteration time between separate benchmarks. As implemented in Murinae, this entire module is no more than sixty-five lines of F# code. This input processor is also adapted and used in the second F# design to be discussed next.

As with other factors external to the propagation engine, input-file processing times are not included in the unit propagation times being measured but are recorded separately. A simple discussion of the performance improvements from this concurrent reader will, however, be included in the results.

4.3.6 F# Variable Locus: Apodemus

Setting aside the two previous solvers, a third solver was developed to explore what unit propagation with two watch literals might look like if it has been developed for this precise problem and environment from the very start. Named for the *European field mouse*, this solver measures our variable-centric design. It also eschews certain language-oriented implementation techniques measured in Murinae that were shown to be less efficient. It has been further extended to illustrate the ease with which introspection can be performed for extended search strategies such as parallel guiding path and portfolio search.

It can be executed from the command-line by specifying an input problem to be solved along with a number of options to be discussed:

$$Apodemus\ [problem.cnf]\ [strategy]\ [node]\ [cores]$$

Search Strategy

As with Murinae and Mus Musculus, the Apodemus solver defaults to a unified search strategy. The solver uses immutable types extensively, however. As a consequence, it can run guiding path or portfolio search with only minimal alteration. Since a snapshot of any intermediate clause database can be safely obtained without interrupting existing search, the portfolio method requires only four additional lines of code to launch any number of competitors. The parallel guiding path technique requires slightly further modification due only to its shared delegatable choice queue. This shared queue is used in lieu of the immutable alternative choice queue employed by the unified and portfolio search strategies. Apodemus accepts a *strategy* parameter on the command-line to specify the desired search technique.

Node Type

Two implementations of variables exist in Apodemus. One is based purely on language integrated asynchronous workflows, whereas the other uses MailboxProcessors in addition to the asynchronous workflows, introduced briefly in Section 4.2.5. From the command line, they appear as *Node* and *Agent* options, respectively. Although both are available, measurements are taken exclusively with the first implementation.

Affinity

Apodemus contains the same options for configurable affinity to dynamically change CPU core assignment as discussed in Sections 4.2.3 and 4.3.4. This can be configured through the *cores* parameter at the command-line with fundamentally the same values as Murinae.

4.4 Measures

A number of measurements are taken during the benchmarking process including both system measurements, to identify how the programs interact with their operating environment, and search measurements, to estimate how well the programs are performing in their overall tasks.

4.4.1 Systemic

System measurements are taken from the Windows Management Instrumentation (WMI) facilities of the operating system. This provides various levels of granularity in its measurements from system-wide performance counters to process and application domain performance counters. This toolset is integrated into all versions of Windows and is a well established method for measuring system performance.

An important such measure is the program's memory footprint or allocation size. This is the $Process \rightarrow Working\ Set$ performance counter. Benchmarks include measurements of the memory consumed at various states of execution. Of initial importance is the memory allocation after the DIMACS input file has been converted into the engine's internal representation but before search artifacts have been constructed around that representation. Subsequent measurements identify the additional consumption as a consequence of search operations including intermediate incremental state and growth of learned information as search progresses.

In addition to the allocation, specialized measurements of system-wide cache behavior are also taken. For these, several iterations provide assurance that other system-level processing is not influencing the measurements. Specifically, cache faults are of particular interest to the performance of solvers since they indicate that a pause in execution was necessary in order to obtain additional information from main memory. In particular, the performance counter discussed in Section 5.2.2 is $Memory \rightarrow Cache\ Faults/s$.

4.4.2 Search

Three important search measurements are used to estimate the overall performance of these engines. In particular, the total number of units propagated, the number of clauses propagated, and the number of conflicts encountered. The number of units propagated gives an immediate indication of the efficiency of the unit propagation engine, regardless of how many clauses are visited on dispatch for each watch list. The number of clauses propagated counts the number of independent clauses visited during propagation. With these two values, it is possible to estimate the approximate average length of watch lists in the engine, since this is just the number of clauses propagated per unit:

$$\textit{Watch Length} \approx \textit{Clauses Propagated} \,/\, \textit{Units Propagated}$$

This is an approximate number since unit propagation engines can break traversal of clauses during watch list dispatch by raising a conflict early in the process. When this happens, subsequent propagation to clauses also on the list may be skipped. Nonetheless it is a helpful approximation.

Finally, the number of conflicts encountered provides an additional indication of the propagation engine's ability to traverse the search space, subject to certain constants which will now be discussed.

4.5 Parameters

A large number of factors influence the performance of a propagation engine, and logic program solvers in general. Some of these parameters are intentionally varied during measurement and others are specifically held constant by design.

4.5.1 Constant

Among the factors held constant during measurement, particular attention is paid to otherwise nondeterministic aspects of the solver. For instance, most solvers employ a random component to the decision-making process. More specifically, this plays a role in the choice of assumptions while solving, i.e. which literals to suppose true or false. Heuristics play an important role in this choice, in order to drive search towards highly conflicted areas of the search space. In all cases where a random factor would normally be involved, the software being tested will prefer the first applicable choice. In the case of literals or variables, this choice will be ordinal according to their numeric value, e.g. (1, 2, 3,...). For clauses, the choice will be made according to the chronology in which they were read or learned. Consequently, all paths taken by the competing unit propagation engines through the search space will be as similar as possible under the circumstances. The intention is to comparatively examine the characteristics of the unit propagation rather than the performance of an overall solver.

4.5.2 Varied

A number of facets of the unit propagation procedures are varied parametrically through either compile-time or command-line options. For instance, the data type and corresponding algorithms of Murinae are varied between either mutable or immutable types depending on the command-line option selected. Apodemus is a predominantly immutable design, whereas the C# reference design is a purely mutable implementation. Threading is implicit in all designs except Murinae which supports both implicit and explicit experimentally. Varying between the two threading options is intended to examine aspects of memory locality and partitioning. Both F# designs accept command-line options for CPU affinity, which are measured for all combinations up to the available sixteen cores. Finally, Apodemus supports three different search strategies, unified, portfolio and guiding path splitting, with measurements being taken in all three scenarios. Murinae and Mus Musculus are both designed for a single unified search.

The only compile-time option varied between all three solvers is the word size. Results varied substantially depending on whether the programs were compiled for 32bit or 64bit execution, so measurements of both were taken.

4.6 Tools for Analysis

A variety of tools are used for analysis of the measurements taken during benchmarking. The output format of the propagation engines is configured for comma-separated values which can easily be imported into Microsoft SQL Server 2008 R2 databases by way of Excel spreadsheets. Once in structured relational form, custom presentations of the data are prepared primarily in Crystal Reports 2008 with additional analysis from Matlab R2010a.

Beyond just the measurements taken during benchmarking, RedGate ANTS Memory Profiler 5 is employed to access details of memory usage and runtime heap allocations. During development, ANTS Performance Profiler 5 was used to identify and correct several limitations of the specific implementations we tested.

Finally, the concurrency profiling tools included with Visual Studio 2010 Ultimate became available very late in the research process. Results from this tool were useful to verify aspects of concurrent execution, however Visual Studio 2010 Professional is the development platform for all tested versions of our software.

Chapter 5

Measurement

5.1 Discussion of Murinae

The benchmark results from Murinae serve two fundamental purposes. On one hand, aspects of the language and its environment are being measured to better understand how they relate to the solver's overall performance. On the other, its clause-locus design is being measured to compare with both Mus Musculus, a conventional serial design, and Apodemus the subsequent implementation based on a variable-locus.

Performance information learned from the various configurations of Murinae within the F# environment will be used in the Apodemus implementation in order to better focus its measurements on domain specific problems.

5.1.1 vs. Mus Musculus

As an important initial sanity check, the clause-oriented F# solver is compared with the C# implementation. Its average unit propagation rate across the full suite of benchmarks is illustrated in Figure 5.1. Although the clause-oriented functional design is outperformed by the C# solver, these measurements assume single-core affinity. The reduced performance in this scenario can be partially attributed to the overhead cost of concurrent execution within a single-core environment.

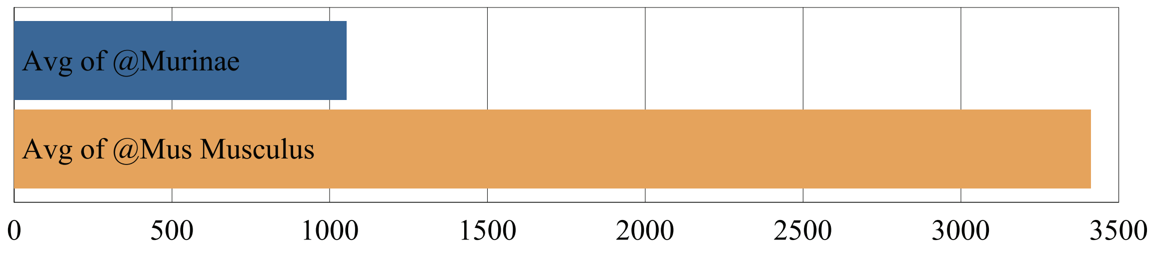

Figure 5.1: Unit Propagations Per Second (Single-core Murinae)

Although Mus Musculus, the C# implementation, performs well here, it does not outperform Murinae, our first F# solver, in all scenarios. The clause-oriented F# solver measures substantially better on cryptographic problems such as the AES key-search benchmark results depicted in Figure 5.2. These benchmarks have an unusually high rate of clause participation per literal, which is indicative longer watch lists. Listed in Table A.1, this specific benchmark also has the highest average clause length among the input logic programs used.

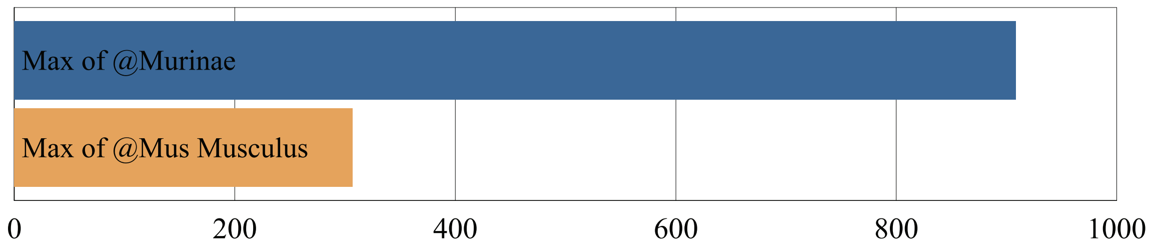

Figure 5.2: $AES\_128\_10\_KeyFind\_1.cnf$

Having demonstrated that the Murinae solver is loosely within the same order of magnitude as Mus Musculus, a more in depth analysis of its underlying components will help to further understand its performance.

5.1.2 Threading Techniques and Memory Locality

Among the components varied at runtime, two concurrency techniques are measured to learn whether an explicit or implicit implementation of multi-threading is desirable. The benefits of language-integrated concurrency are compelling, but the potential for higher memory locality with an explicit handling of asynchronous processing needs to be explored.

The outcome of these results in the general case is fairly clear. The language-integrated, implicit asynchrony outperforms the explicit implementation in every scenario as summarized in Figure 5.3.

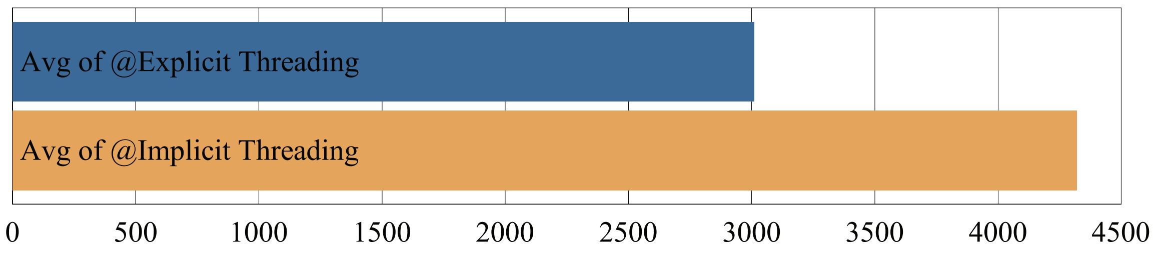

Figure 5.3: Implicit vs. Explicit Threading (Multi-core Murinae)

A conclusion of this result is that the work-stealing, duplicating queues used by the language work better here given the way clause dispatch is performed. Notably, any benefit to explicitly managed data locality in the allocation and use of Murinae's clause database is outweighed by the improvements in implicit work dispatch. In its paper on the subject, Microsoft Research discusses the implementation of these queues in some detail. In Section 6.3 of this paper, the authors obtain a quad-core speedup with duplicating queues of 3.89, at nearly five-hundred tasks per millisecond, as opposed to the THE protocol which exhibited a speedup of only 2.78 during concurrent execution [24]. Their ratio of $2.78 : 3.89$ is roughly 0.71. We observe a similar ratio with the duplicating queues of $3k : 4.3k$, or just over 0.69, as compared with non-blocking lambda queues illustrated here. As observed in this ratio, the slight improvement in non-blocking queue performance may be due to locality imposed in the underlying structure of the clause database, however the difference is negligible in practice.

Jumping ahead to look at the full set of results including Mus Musculus, Murinae and Apodemus, this picture is further reinforced. A summary of the best solvers for each benchmark demonstrates that the explicit-threading technique is not among the configurations used by any of the three solvers when achieving top scores for the selected benchmarks.

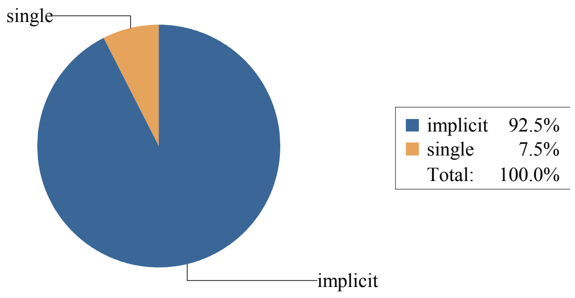

Figure 5.4: Best Scores - Threading Proportions

It was further observed during our testing with WMI, introduced in Section 4.4.1, that the Windows scheduler frequently moved the process to execute on cores that are closer to certain banks of memory in the system architecture. We suspect this behavior is less costly during execution than moving data across the HyperTransport bus to the currently executing process. Although this cannot be verified, it may also play a potential role in these results.

The language-integrated features for dispatching concurrent work outperform the explicit management techniques used in Murinae despite the potential advantages to locality from partitioning the database.

5.1.3 Mutable vs. Immutable Procedures

The second key aspect of the F# language, as tested by Murinae, is the representation of its data, including clauses, watches and variables. Traditional solvers like MiniSAT, whose source code was studied extensively during the course of this research, use mutable data structures to edit watches in place during watch management. Unit propagation algorithms can be easily adapted from these competition solvers to use either the mutable data structures for which they are intended, or immutable data structures. As discussed in Section 4.2.1, mutable operations on data can be easily altered to instead produce new instances of that data with the operation applied.

It is easy to imagine that this approach may not lead to optimal performance for at least two reasons. First, the previous mutation of any instance in memory now involves at least two: the source instance being read and the new instance being created. Second, the frequent recycling of these instances may involve unpredictable penalties for garbage collection and allocation. These perceived shortcomings are supported quite comfortably by the benchmark results illustrated in Figure 5.5.

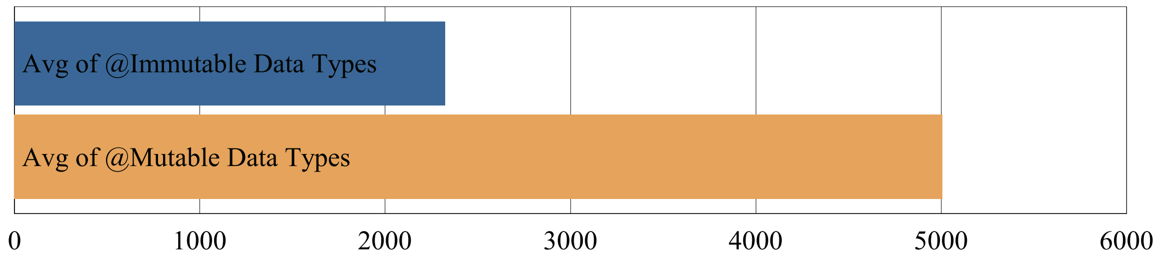

Figure 5.5: Mutable vs. Immutable Data-Types (Multi-core Murinae)

In the results from our benchmarks, it becomes clear quite quickly that the use of immutable data types with existing algorithms is costly. Although we discussed two reasons why the immutable procedures might perform worse, behind these justifications is an important fact: an algorithm that relies on continuous, in-place mutation has been applied to immutable data. This observation is a key impetus to the design and implementation of variable-centric Apodemus.

5.1.4 Scalability

In spite of the previous results, immutable data types might still be compelling if the benefits for concurrent processing outweigh the initial costs of immutability. Figure 5.6 details the scalability characteristics of the Murinae solver as the number of allocated CPU cores is increased. Measurements are grouped according to both the types of data representation and the threading models. These are averaged across the complete series of benchmarks.

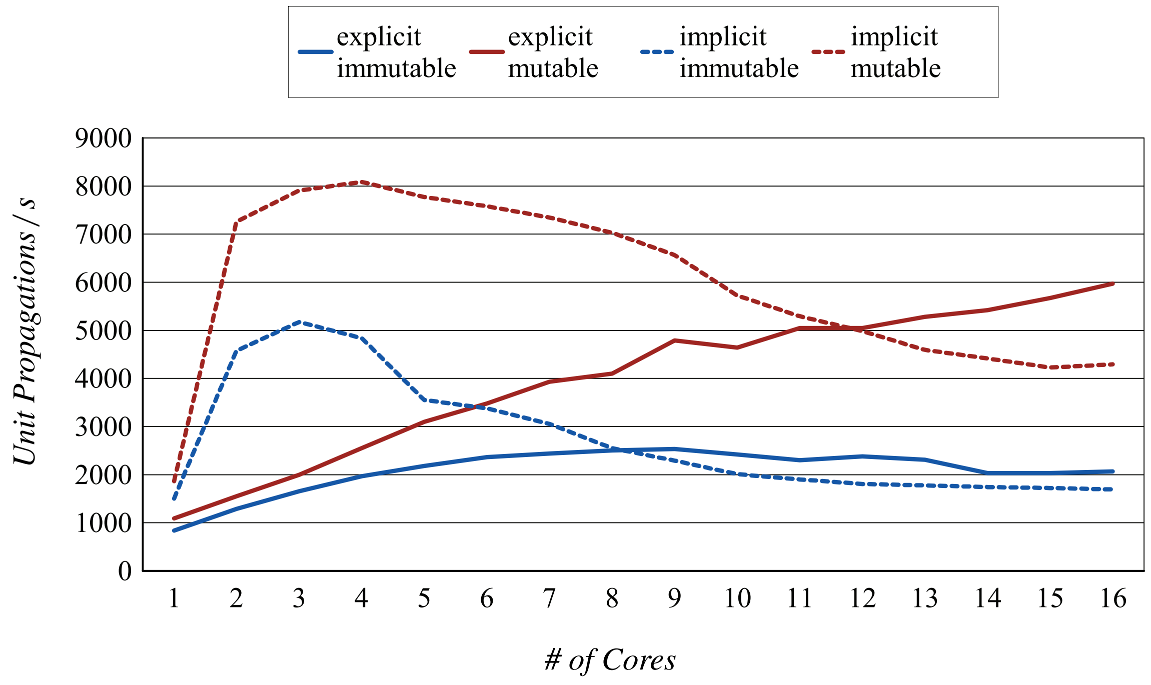

Figure 5.6: Summary of 64 Bit Murinae

This scalability chart for Murinae shows an overall score in terms of unit propagation speed. The efficiency of implicit threading still outmatches explicit work dispatch across the board, with the cost of additional concurrency outweighing its benefits past three and four allocated cores. Recall that this is partly attributable to watch list length and partly influenced by the four available channels to memory. The mutability curves generally follow each other, but in both cases the scalability of immutable data - as applied in Murinae - peaks earlier and diminishes thereafter.

These summarized distribution curves remain consistent between both 32bit and 64bit solvers, although individual benchmark results vary. A chart for 32bit Murinae is included for examination in Figure B.1 of the appendix.

5.1.5 Parallel DIMACS Processing

In order to maximize the timeliness of benchmark runs, Murinae introduced a concurrent DIMACS processor, discussed briefly in Section 4.3.5, to translate input CNF files into internal data structures. To our knowledge this hasn't been done, so performance measurements were taken for inclusion in this work.

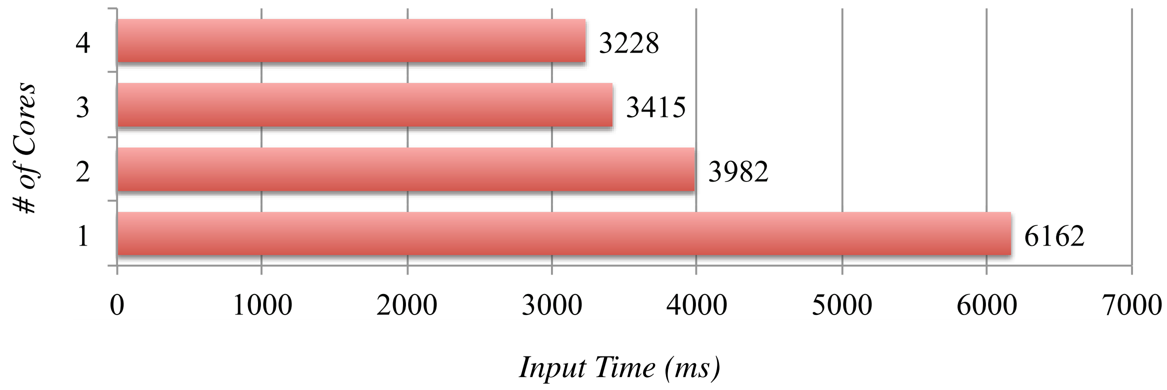

Figure 5.7: Parallel DIMACS Processing

Using language integrated features for concurrent processing of input data, its scalability exhibits results that are similar to the implicit threading techniques previously measured for this architecture. A summary of this performance gain is illustrated in Figure 5.7. Generally speaking, the larger the input problem the greater the benefit, with small problems seeing no benefit from the additional compute power. This observation is supported by detailed measurements and problem size appearing in Table B.1 of the appendix.

5.1.6 Summary of Murinae

Murinae enabled a high degree of experimentation with the language and environment in order to learn how the clause-locus - a natural parallel extension of existing unit propagation procedures - performs in a multi-core system. Several conclusions were drawn about threading, data types and their interactions. Under these usage scenarios, language integrated features for parallelism outperform explicit techniques for watch management in spite of the possibility for improved locality. Moreover, the highly mutable clause-locus procedures give clear preference to mutable data types in virtually all scenarios. Finally, a concurrent input processor was developed to make use of additional unused CPU cores during program initialization.

In spite of these results, important questions were raised. Among these, we recall that the immutable procedure for watch management in the clause locus under-performs in part because that algorithm is designed to operate in-place on mutable data. It becomes important to ask, what would a concurrent unit propagation procedure look like for immutable data? In order to address this during our subsequent work, it became necessary to reposition the locus of concurrent execution from the watched clause and the clause database to the variables under assignment.

5.2 Discussion of Apodemus

If the rate of mutation can be altered, or even largely eliminated in the case of the proposed variable locus, then the benefits of immutable processing to concurrent execution become significant.

Designed to optimize memory behavior in this scenario, the performance of the Apodemus engine is compared head-to-head with Murinae and the C# solver, Mus Musculus, using measurements obtained across the complete set of benchmarks.

5.2.1 vs. Murinae

When each benchmark is measured by both Murinae and Apodemus, we observe a dramatic shift among top scoring solvers for each problem, illustrated in Figure 5.8. Recall that mutable versions of Murinae won in every case, whereas Apodemus is designed specifically for immutable data-types.

A variety of important factors are implemented differently in Apodemus. Although Murinae optionally uses immutable data types for its watched clause database, its search is still fundamentally mutable. When backtracking, for instance, its watches retain their new configuration, whereas Apodemus uses a recursive clause rewriting approach, outlined in Section 3.4.4. As a consequence, Apodemus never needs to alter its clauses, but simply moves its references step-wise inward, as specified in Sections 3.4.1 and 3.4.2. Due to this

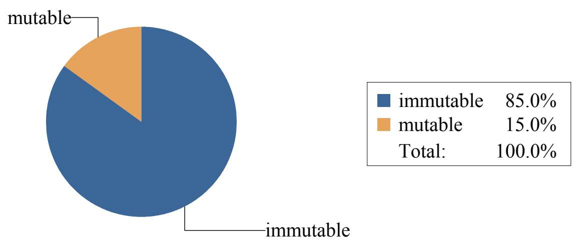

Figure 5.8: Best Scores - Data Type Proportions

sliding window approach, issues of garbage collection and allocation are constrained to references held by the variables rather than the continuously expanding clause database.

5.2.2 Memory

32 and 64 Bit Performance

This reliance on references to intermediate *cons* cells of the clause remainder makes Apodemus significantly more sensitive to the size of its references. In contrast, the size of literals, where they appear as integers in clauses, plays little to no role in the measured results. References that are used by these immutable data structures, however, cannot be changed except by recompiling the program, and their size has a significant influence over the outcome.

Between the 32 and 64 bit builds of Apodemus, an examination of which revision performs best for each benchmark problem yields an interesting result. Illustrated in Figure 5.9, individual benchmarks are plotted according to their size in the number of variables and clauses. This logarithmic scatter plot clearly exposes the divide between the two. Problems which are solved more easily on 32 bit Apodemus appear in purple, whereas those that prefer 64 bit Apodemus appear in green.

In general, smaller problems perform better on the 32 bit version than larger problems. Notably, all instances are capable of executing within the allowed program size for a 32 bit program, eliminating this as a possible bias. Exceptional outliers depicted in the profile of

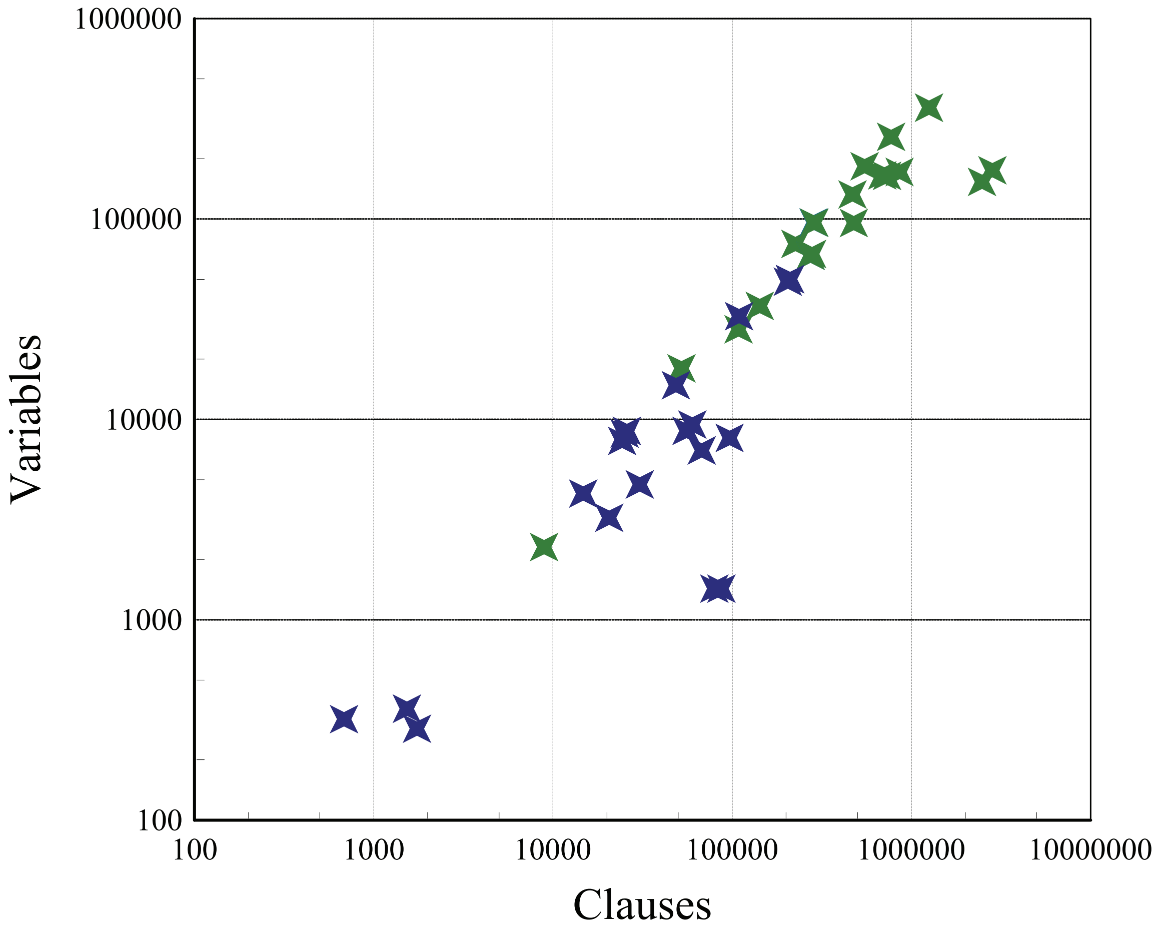

Figure 5.9: Best Solver - 32 vs. 64 Bit

initial benchmark problems, including Figure A.1 and Table A.1 of the appendix, are not included here.

Normally, execution of native 64 bit instructions will be more responsive, and this is apparent from many of the larger benchmark problems. In this case, however, we believe that the 32 bit configuration is the only one in which the entire logic program fits within cache given a sufficiently small problem. To support this, we examine the largest group of problems successfully solved by 32 bit Apodemus: the *mizh-sha* benchmarks listed in Table B.2. Their profiles in Table A.1 indicate that these problems consist of approximately 210,000 clauses with an average length of 3 literals spread across nearly 50,000 variables. Since their representation in memory consists of *cons* cells containing an integer for the literal and a reference to the next list element, we obtain approximately 630,000 words for literals and an additional 630,000 for references. Moreover, the lists exist in both directions, forward and reverse, so these figures are doubled. These 2.5 million words compose the immutable clause database itself. As described in Section 4.1.2, the AMD Opteron 6128 CPU has 12 MB of shared L3 cache, approximately 3.1 million 32 bit words. The remaining 600,000 or so can potentially be used to capture the intermediate remainders referenced by variable watch lists. To obtain a unit or conflict, the watch list would traverse on average half of a clause with its reverse traversing half in the opposite direction. Assuming an average length of 3 literals per clause, a unit can be obtained by moving the forward or reverse lists a total of 2 positions inward. This intermediate data accounts for an additional 420,000 words in the worst case. The operating system, .NET framework, and the program itself could easily occupy portions of this space as well. As suggested by these results, the 32 bit representation is the only scenario where the entire problem and its structures fits within the available shared cache. A native 64 bit version operating entirely in cache would require problems of roughly half the size in order to compete successfully on level ground.

Initial Allocation Size

Single-core Apodemus and Murinae exhibit similar initial memory allocations despite their substantially different configurations. The initial memory allocations for a sampling of benchmark problems is listed in Table 5.1 to illustrate this similarity.

Immutable designs, however, have another significant advantage over mutable ones for concurrent processing. Apodemus easily implemented three different search strategies that are described in Section 4.3.6: portfolio, guiding path splitting and unified search. Since it was known that no worker could unintentionally alter the initial problem, it could be shared among all concurrent workers. To illustrate the potential up-front cost, an estimated average memory allocation for sixteen competitors is placed alongside the measured best allocation size of single-core Murinae for comparison with Apodemus. Unlike the incremental memory acquisition of immutable data, this must be populated up front for each competitor without regard for the solving time of a problem. For trivial problems on systems with more than sixteen cores, this can potentially be quite costly to the critical-path solving time.

Benchmark	Apodemus	Murinae	x16
aes_128_10_keyfind_1.cnf	113	107	1,830
aloul-chnl11-13.cnf	19	20	337
am_7_7.shuffled-as.sat03-363.cnf	28	30	488
cmu-bmc-longmult15.cnf	34	36	577
countbitsarray04_32.cnf	35	38	592
goldb-heqc-alu4mul.cnf	40	43	677
li-test4-100.shuffled-as.sat03-370.cnf	137	165	2,606
manol-pipe-c6bidw_i.cnf	201	244	3,730
rbcl_xits_09_unknown.cnf	67	69	1,139
rpoc_xits_09_unsat.cnf	73	77	1,201
satisfiable.cnf	17	20	321
simon-s03-w08-15.cnf	331	496	8,629
slp-synthesis-aes-bottom17.cnf	90	96	1,555
smtlib-qfbv-aigs-countbits128-tseitin.cnf	206	215	3,531
velev-engi-uns-1.0-4nd.cnf	61	69	1,085

Table 5.1: Portfolio Allocation Sizes (Mb)

The impact on shared memory cache for modern multi-core architectures is also an important outcome of this measurement. Recall that the Opteron 6128 CPU, described in Section 4.1.2 covering the benchmark platform, shares 12MB of L3 cache between its eight cores. For the twin CPU arrangement used to benchmark this work, there are only two large caches of this type. In this example, large portions of the shared clause database on Apodemus can easily be cached, since all references to clause remainders will point to addresses within this root program. Traditional mutable systems that alter the clause database by moving watched literals to the front of a clause as search progresses will suffer more frequent cache faults as each of their sixteen cores require access to different areas of memory — a problem to be discussed further in Section 5.2.2.

Progressive Allocation Size

A potential disadvantage of immutability here, is that differential allocation from the base program takes place during solving and thus represents an ongoing cost. Moreover, this yields a memory footprint that is no longer purely linear in the number of clauses. Instead, the depth of search and the degree to which it has applied incremental changes from the base program also factor into the ongoing allocation requirements.

To illustrate the non-linear memory requirements, the intermediate-sized benchmark *goldb-heqc-alu4mul.cnf* has been measured with single-core affinity over the course of a thirty-minute execution and plotted in Figure 5.10. This figure captures the total allocated memory as seen from the operating system for a one-to-one comparison of the solvers. In addition to Apodemus, both the mutable and immutable variants of Murinae are included for comparison.

Visible in these measurements is the gen-2 garbage collection which takes place at evenly spaced intervals just over four minutes apart. In the current version of the .net framework, collection of gen-0 and gen-1 memory occurs on an ongoing basis whereas the gen-2 collection exhibits this pattern of behavior. For purposes of reproducing these measurements in any future work, it should be noted that the gen-2 garbage collection will be revised in the upcoming .net framework version 4.5 to enable concurrent background operation.

At first glance, these measurements might seem unnerving when compared with a traditional competition solver, however two important factors need to be considered. First, a conflict-driven, clause-learning solver continuously prunes its database as introduced in Section 1.4. As a consequence of this optimizing behavior, the total allocated size will typically be only a very small portion of the available system memory. Despite the larger

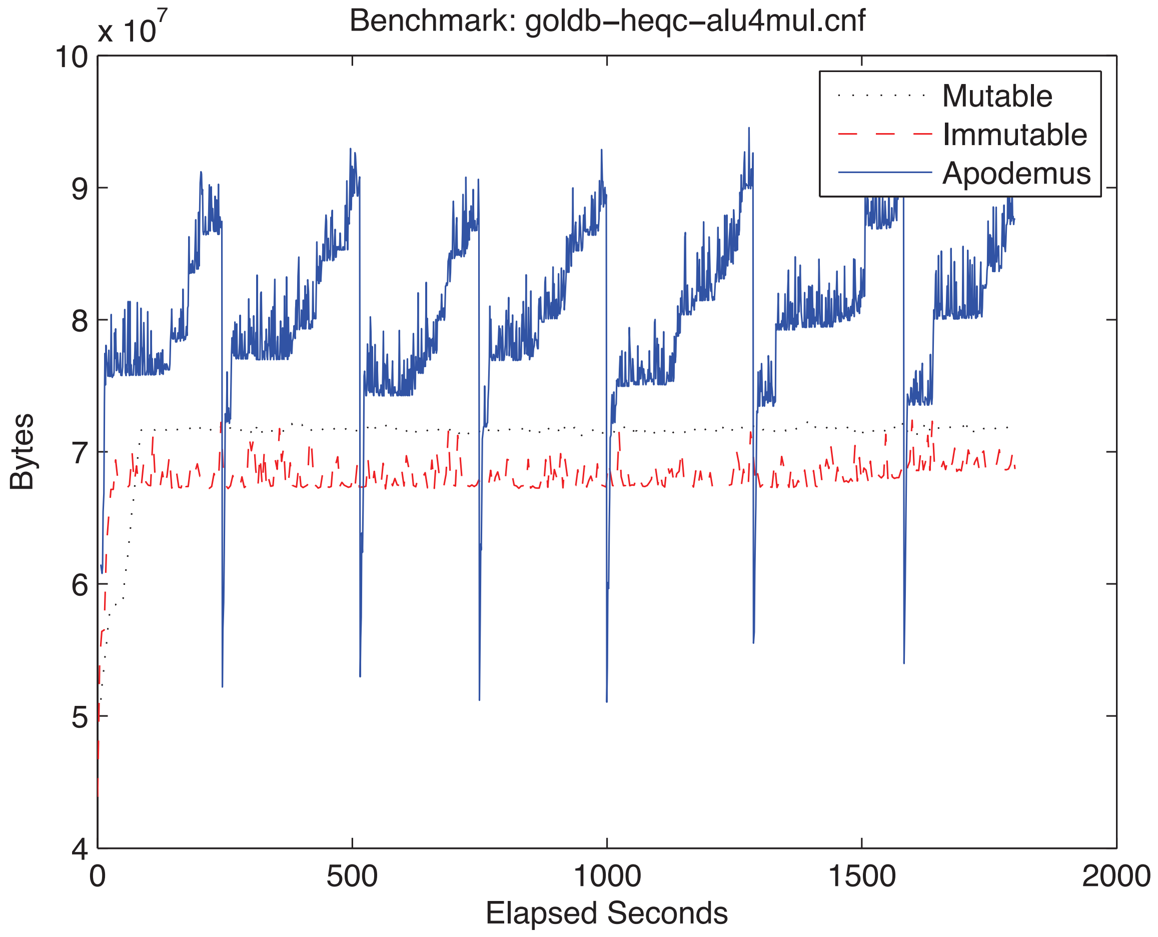

Figure 5.10: Memory Usage - goldb-heqc-alu4mul

and varying allocation sizes, the variable-locus design for immutable types still only operates on a certain view of its incremental state and thus retains much of the benefits of cache optimization underlying the current approach to clause management, a result which will be discussed next. Second, when compared in single-core mode, the overall performance of the Apodemus unit propagation engine on this benchmark far exceeds that of both Murinae and Mus Musculus when averaged across several runs. This result is depicted in Figure 5.11.

For this particular benchmark, Apodemus runs an average of over thirty-thousand unit propagations per second as compared with just under two-thousand for Murinae and four-

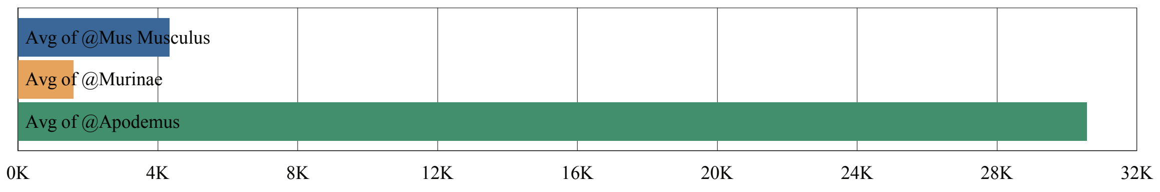

Figure 5.11: Performance Comparison - goldb-heqc-alu4mul

thousand for Mus Musculus, the C# serial solver.

Cache Behavior

To further understand the implications of this cache behavior, extensive measurements were taken of the three solver configurations. Figure 5.12 illustrates the number of cache faults per second as observed in the middle of a thirty-minute run on the *goldb-heqc-alu4mul.cnf* benchmark used in the previous discussions. For this chart, lower numbers of faults are preferred in order to maximize performance.

Once again, Apodemus is compared with the mutable and immutable variants of Murinae. It consistently demonstrates a greater occurrence of snapshots observing zero cache faults. The Murinae engine, however, frequently misses and must subsequently obtain that information from system memory. As mentioned in Sections 2.1.1 and 4.1.2, the Opteron system used for benchmarking has only four channels to main memory and excessive requests from multiple CPU cores can easily bottleneck here.

As the number of competitors in a portfolio or guiding path search increases, the overall cost of this problem will magnify. This will become more apparent in Section 5.3 which looks at the overall multi-core throughput across the complete set of benchmarks.

Over the total thirty minutes of execution, Apodemus snapshots an average of only 1.26 cache faults per second compared with 2.37 faults per second under Murinae. For this particular execution, immutable Murinae performed slightly better than its mutable algorithms, potentially due to the locality of its newly modified watches rather than an arrangement imposed by the order in which clauses are read or learned — although this

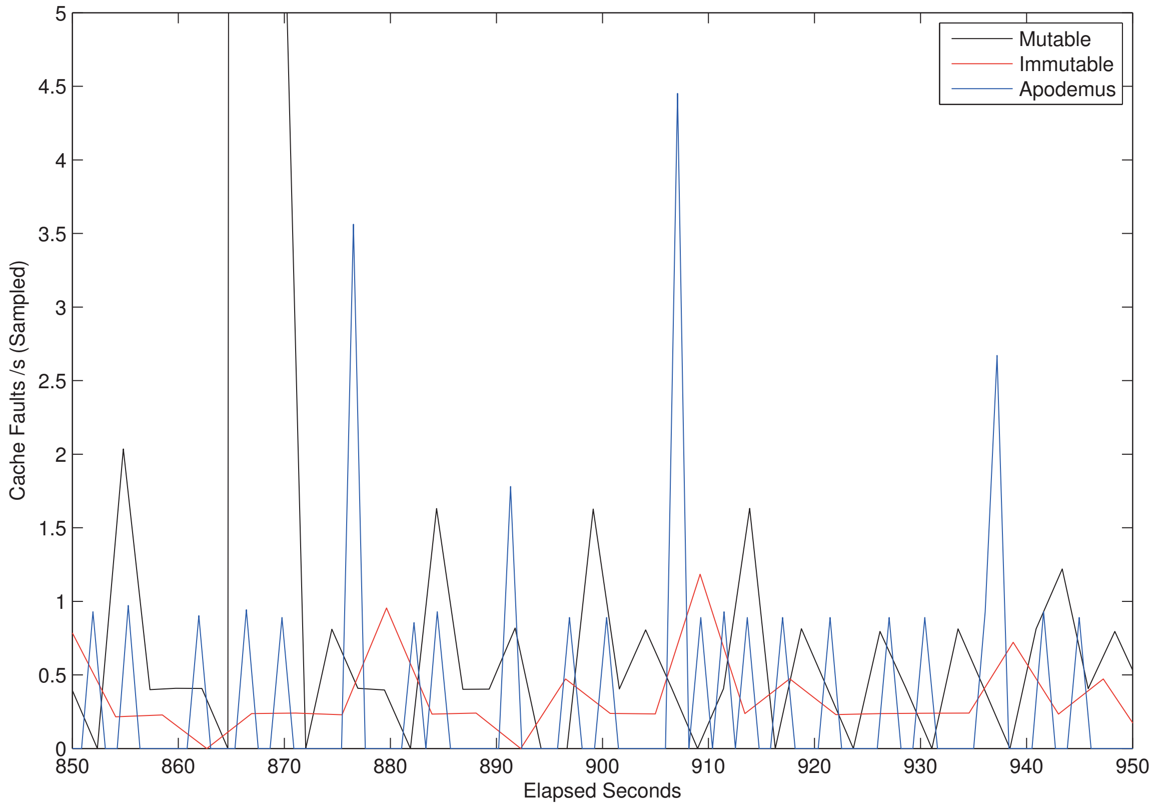

Figure 5.12: Cache Faults - goldb-heqc-alu4mul

speculation is unsupported.

5.2.3 Summary of Apodemus

As a solver, Apodemus shows potential. Its variable-locus implementation addresses certain control issues that enable some compelling concurrent capabilities, including advanced search strategies and speculative unit propagation. Its watched clause rewriting pairs elegantly with immutable types to facilitate state introspection. Overall, its characteristics are different from Murinae and many other conventional solvers. Its sensitivity to word size and increased allocation in deep search is balanced against its modest initial allocation size. Concurrency and cache behavior perform reasonably and its program length is comparatively smaller than Murinae. Fundamentally at issue is its memory performance and whether its unit propagation engine delivers. A more in depth look at overall throughput compared with the other two solvers will now analyze this aspect of performance.

5.3 Overall Throughput

Most of the measurements discussed so far have focused on specific and narrow aspects of the comparative performance among the solvers being tested. It is especially interesting, now, to evaluate the overall performance of the three solvers across the complete set of benchmarks using anything up to sixteen cores and including all configurable options under measurement. These include data types, threading models, word size and search strategies.

Results from the full set of over thirteen thousand measured executions are included in this analysis. In particular, we look at which solver configurations are most successful for each problem. Detailed results of this comparison are included in Table B.2 of the Appendix for reference during this discussion. Entries in Table B.2 are sorted according to increasing problem size in the number of variables.

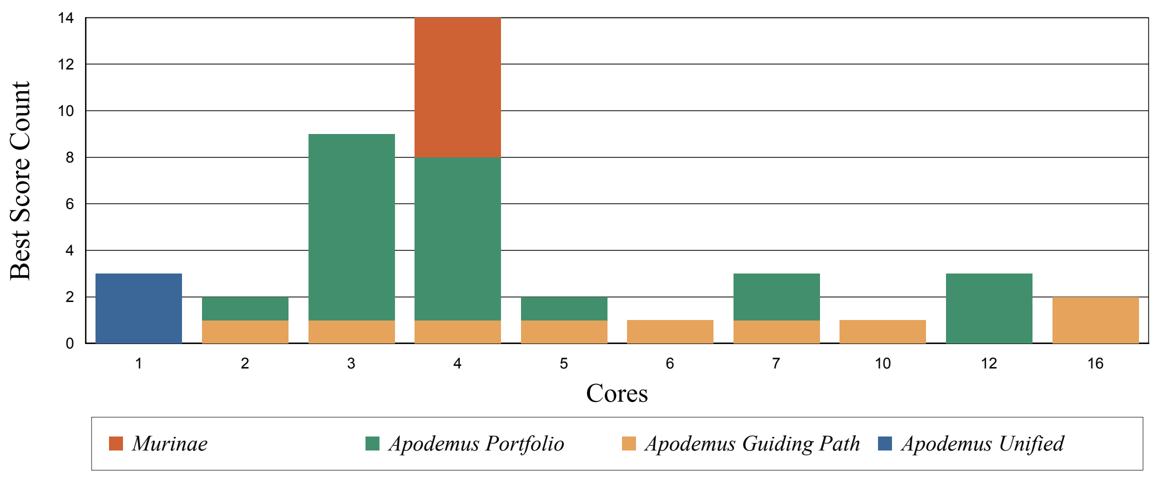

Figure 5.13: Best Score Counts - By Solver and Cores

An important aim of this work has been to improve the performance of multi-core unit propagation using functional language constructs. When reviewing the results for Murinae in Section 5.1.4, it becomes clear that its best performing configurations peak when assigned affinity between three and four cores. This result is especially true when competing with the other two solvers.

In Figure 5.13, Murinae with its clause locus achieves top propagation performance for benchmarks when it is using exactly four cores. In all of these scenarios, Murinae is configured for unified search, mutable data-types and implicit threading. Other combinations are outperformed either by this configuration or some configuration of the Apodemus solver.

Some problems may contain highly sequential structure in their implication graphs. For these, Murinae will be inefficient due to its concurrent watch dispatch. Mus Musculus, the C# solver, or Apodemus are the only likely competitors here. In this case, unified search on Apodemus with single-core affinity produces higher throughput.

For the remainder of problems, guiding path and portfolio search share as top performers with their configurations. It is interesting to observe that guiding path splitting has a more right-ward skew in Figure 5.9. Although portfolio search is an area of intensive ongoing work, advocates of parallel guiding path search may have reason to remain optimistic. Guiding path splitting is a strategy favored by parallel solvers like PaMira and strongly advocated by its authors [12].

5.4 Final Considerations

Among measurements taken during development, it was observed that CPU utilization depended primarily on the efficiency of memory access, especially with unrestricted affinity. As a consequence of its memory efficient design, Apodemus demonstrates a clause structure which performs well for this reason.

Evident in these results, however, no single center of control is clearly or inherently better than another but each has its place across a spectrum of diverse problems. Nonetheless, the variable-centric design used by Apodemus, with its cache-friendly clause representation and immutable state, enables the highest unit propagation rates on eighty-five percent of the benchmark files tested. Although this work experimentally tests clause and variable locus propagation designs, other centers of control may be identified, at least one of which will be discussed shortly in Section 6.2.1.

Chapter 6

Future Research

6.1 Watch Dispersion

An important assumption of the immutably-typed variable-locus employed in Apodemus is that watch references will be moved step-wise inward on the forward and reverse remainders. A consequence of this, clauses prefer initial placement on watch lists in precisely the order in which these literals appear. A study of watch dispersion patterns in this locus could distinctively alter the runtime behavior by favoring the front-loading or back-loading of common literals in this selection procedure. More sophisticated work could evaluate uniform and non-uniform dispersion to determine whether immutable unit propagation performance is improved in either scenario given current heuristics and search strategies.

6.2 Enclosed Two-Watch Literal Inversion

6.2.1 Inversion

One of the most problematic issues for a monolithic clause-oriented design is that non-trivial clauses appear simultaneously on two watch lists. This poses a problem for any operation which may concurrently alter both lists. A desire to avoid blocking data structures and this duality of control lead us to try a language-specific solution. In F#, the tuple data-structure provides an alternative single point-of-control which contains both current watch literals for a clause, such as (w_1, w_2). Since the number of literals in most solvers is both finite and unchanging, the number of watch lists can be approximately squared. The solver would then use lazy concurrent data structures to generate independent watch lists for each pairing of literals. As a consequence, all clauses with both active watches on w_1 and w_2 would appear on the watch list for tuple (w_1, w_2).

In this model, any given clause is on exactly one watch list at a time. Due to this locus, the scope of propagation for some literal is the set of all non-empty watch lists in which that literal appears. By imposing a natural ordering to the tuple and using indexing, we simplified the process of visiting all watch lists containing a given literal. Despite these improvements, the sparseness of literal pairs resulted in excessive time spent enumerating a literal's watch lists.

6.2.2 Enclosure

If the disadvantages of enumerating sparse watch pairs can be addressed, it would be possible to develop a literal-free propagation engine built around translation between input clauses under closure and the procedures which handle watch management. Function composition and an immutable model similar to the variable-oriented design presented herein offer a compelling and natural structure for a functional solver.

The conceptual model is more akin to lambda calculus than contemporary data-oriented functional languages and would present an interesting theoretical assembly. In particular, the contents of watch lists would contain lambda functions enclosing the deterministic consequences of invalidating that watched pair. Moreover, if such an assembly could be constructed as a translation of the raw DIMACS formatted CNF file, the integer data structures of today's solvers could be forgone entirely.

6.3 Portfolio-Based Heuristic Phasing

A critical impetus to the work on immutability, concurrency and unit propagation for our research group is our prior experimentation with phased clause exchange. Based on game theory and international economics, the notion is that competitors in a parallel portfolio search can exchange clauses the same way nations exchange goods.

Our work in this area focused around biases to the selection heuristics used by CDCL solvers to make assumptions and to identify highly conflicted space. By developing heuristic biases for the production, exchange and consumption of clauses, a collection of portfolio competitors could generate synergy through specialization in certain aspects of the problem. Rather than attempting to assign some value to an arbitrary clause, the clause itself becomes valuable, in a sense, given the search environment and bias under which it was created. Clauses which prove useful under exchange, then, can be used as feedback to the heuristics of both the producer and consumer.

In the same way that two distant countries producing the same goods will be less likely to engage in trade, the enabled degree of phase between competitors is, in part, a function of the latency between them. For such a strategy to work, however, two important concurrency problems have to be solved. First, the frequency between sender and receiver must be adequately small, so that the receiver will not have had sufficient time to generate the same clause in spite of its biases. Second, the introspection of solver state, notably its clause database must not extensively interrupt the competitor from its primary task of searching for solutions. In our experimentation, increasing the frequency of communication between sender and receiver crippled the conventional solver's ability to carry out its primary task. This deficiency is an important impetus for the research presented herein. With additional work, this strategy could be retried given the introspection capabilities of immutable data structures.

6.4 Application To Competition Solvers

Finally, the long term goal is to improve the state-of-the-art in real-world logic problem solving software. Whether this entails improving functional language solvers to the point that overhead penalties of their language and environment are outweighed by the benefits of executing in that environment, or whether it involves adapting the patterns of this software to traditional C-type languages remains to be seen. In the near term, adapting a conventional competition solver with design patterns from this work seems a reasonable approach, but given adequate sophistication, the advantages of a managed functional environment for competition solving are compelling.

Chapter 7

Summary

Over the course of this thesis, we have introduced the domain of answer set programming and discussed aspects of its shared ancestry with satisfiability solving. Both implement derivatives of the Davis-Putnam-Logemann-Loveland procedure to power their search. DPLL offers unit propagation as a technique to derive the consequences of an assignment according to the structure of an underlying problem. The modern two-literal watch scheme is then introduced in order to present a more efficient method for detecting these units. Next, the structure of CNF satisfiability problems is defined, along with some key benefits to early recursive implementations. Conflict-driven, clause-learning techniques developed within the last decade, which are shared between the two logic programming paradigms, are then presented. These advances, including heuristic search and conflict clause generation using UIP, empower solvers to bypass areas of search that DPLL would otherwise explore. Instead, search is focused on highly conflicted areas of the problem space. Finally, popular parallel coordination strategies, such as guiding path and portfolio search, enable solvers to operate in today's high performance computing environments.

In light of emergent languages and multi-core hardware, however, the motivating impetus for this research is presented. Systems architecture and, in particular, memory contention combine with a desire to enable more advanced runtime capabilities. Together, they raise interest in alternative languages and patterns, such as concurrent F# and immutable types. In order to explore these opportunities, the most costly aspect of modern solvers is selected as a focus of research: the unit propagation engine.

After studying the propagation engines of modern solvers such as MiniSAT, it became apparent that implementing concurrent capabilities within these engines was problematic for a myriad of reasons. Careful analysis of the underlying procedure and its component parts identified at least two strategies for implementing parallelism. As a consequence, two polar techniques are proposed within this work. The first focuses on a natural extension of the existing mutable algorithms, treating clauses that require watch maintenance as independent tasks. In contrast, the second revisits conventional algorithms to enable immutability within an alternative center of control. In this case, all operations on variables of the problem are treated as separate tasks encompassing the clause remainders they control. To test these models, three solvers are developed for the Microsoft Windows platform.

The first, Mus Musculus, is a CDCL solver written in C# to provide a sanity check for subsequent work in F#. It establishes an important baseline with which subsequent work can be compared, given the shared architecture of the .NET Framework's virtual machine and libraries.

The second solver, Murinae, implements the clause locus for concurrent propagation and further enables certain aspects of the F# runtime to be studied in order to narrow the focus of subsequent work. It also demonstrates the limitations of the mutable unit propagation procedure as it applies to immutable data types. Moreover, it establishes the preferred implementation of threading behavior within F# as it pertains to management of the clause database. As a unique addition, Murinae introduces a parallel DIMACS processor to reduce input times. Due to its effectiveness, this feature is then reused in the third solver. This processor saw meaningful reductions in critical-path time for large problems with no measurable penalty on trivial benchmarks.

Finally, the Apodemus solver is an implementation of the variable locus. It is developed to gauge the strengths and weaknesses of the proposed design. Due to its extensive use of immutable types, it enables parallel portfolio search with merely four additional lines of code. Moreover, its guiding path search branches intermediate state from the clause database on an ongoing basis with a similarly small overhead to introspection. The cache behavior exhibited by Apodemus is also compelling, yielding a lower overall rate of cache faults on real world problems while simultaneously improving concurrent propagation rates.

Over thirteen thousand benchmark executions, across a series of well-known satisfiability problems, measure various aspects of these solvers. In particular, the unit propagation rates and memory behaviors play an important role in the top performing configurations. Although evaluating the results showed no clear winner, the variable-locus solver exhibits compelling performance both in concurrent execution and cache behavior. The clause-locus design found itself constrained to very specific configurations and outperformed Apodemus on only fifteen percent of the selected problems. In contrast, the variable-locus proved itself to be more versatile, measuring top propagation rates across a full range of scenarios.

Several ideas for future work are then suggested that raise important implications of this research, including watch dispersion, two-watch literal inversion, and portfolio-based heuristic phasing. The benefits of an immutable, concurrent design enable new areas for research that were previously unreachable. With the hope of delivering positive results to the logic programming community, future work may apply aspects of this research to enable high-performance competition solvers to carry out advanced introspection of their state without impeding ongoing search.

Bibliography

[1] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. A User's Guide to gringo, clasp, clingo, and iclingo. Preliminary Draft, 2010. Available at URL[1].

[2] DIMACS Challenge — Satisfiability: Suggested Format, 1993. Available at URL[2].

[3] M. Davis, G. Logemann, and D. W. Loveland. A Machine Program for Theorem-Proving. *Commun. ACM*, 5(7):394–397, 1962.

[4] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, 1960.

[5] A. C. Doyle. Adventure of the Blanched Soldier. In *The Case Book of Sherlock Holmes*. John Murray, 1927.

[6] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, pages 530–535, 2001.

[7] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In *ICCAD*, pages 279–285, 2001.

[8] J. P. Marques Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[1] `http://iweb.dl.sourceforge.net/project/potassco/potassco_guide/2010-10-04/guide.pdf`

[2] `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/`

[9] E. Goldberg and Y. Novikov. BerkMin: A fast and robust Sat-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007.

[10] C. Sinz, W. Blochinger, and W. Küchlin. PaSAT - Parallel SAT-Checking with Lemma Exchange: Implementation and Applications. *Electronic Notes in Discrete Mathematics*, 9:205–216, 2001.

[11] G. Audemard and L. Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *IJCAI*, pages 399–404, 2009.

[12] T. Schubert, M. D. T. Lewis, and B. Becker. PaMira - A Parallel SAT Solver with Knowledge Sharing. In *MTV*, pages 29–36, 2005.

[13] J. Gressmann, T. Janhunen, R. E. Mercer, T. Schaub, S. Thiele, and R. Tichy. Platypus: A Platform for Distributed Answer Set Solving. In *LPNMR*, pages 227–239, 2005.

[14] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, and B. Schnor. Cluster-Based ASP Solving with *claspar*. In *LPNMR*, pages 364–369, 2011.

[15] C. P. Gomes, B. Selman, and H. A. Kautz. Boosting Combinatorial Search Through Randomization. In *AAAI/IAAI*, pages 431–437, 1998.

[16] A. Biere. PicoSAT Essentials. *JSAT*, 4(2-4):75–97, 2008.

[17] J. Huang. The Effect of Restarts on the Efficiency of Clause Learning. In *IJCAI*, pages 2318–2323, 2007.

[18] Y. Hamadi, S. Jabbour, and L. Sais. Control-Based Clause Sharing in Parallel SAT Solving. In *IJCAI*, pages 499–504, 2009.

[19] M. Brain and M. De Vos. The Significance of Memory Costs in Answer Set Solver Implementation. *J. Log. Comput.*, 19(4):615–641, 2009.

[20] SAT 2011 Competition: 32 Cores Track: Ranking Of Solvers, 2011. Available at URL[3].

[21] G. Audemard, J.-M. Lagniez, B. Mazure, and L. Sais. On Freezing and Reactivating Learnt Clauses. In *SAT*, pages 188–200, 2011.

[22] D. Syme. The F# Language Specification. Preliminary Draft, 2010. Available at URL[4].

[23] D. Syme, T. Petricek, and D. Lomov. The F# Asynchronous Programming Model. In *Proceedings of Practical Aspects of Declarative Languages*, PADL 2011, 2011.

[24] D. Leijen, W. Schulte, and S. Burckhardt. The Design of a Task Parallel Library. In *OOPSLA*, pages 227–242, 2009.

[25] E. Meijer, R. Wa, and J. Gough. Technical Overview of the Common Language Runtime, 2000.

[26] Threading Building Blocks (TBB), 2012. Available at URL[5].

[27] I. A. Lee and Z. Zhang. Parallelizing MiniSat. MIT 6.884 Final Project Report, 2010. Available at URL[6].

[28] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*, pages 212–223, 1998.

[29] C. E. Leiserson. The Cilk++ Concurrency Platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[30] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin Berlin. Reducers and Other Cilk++ Hyperobjects. In *SPAA*, pages 79–90, 2009.

[3] http://www.cril.univ-artois.fr/SAT11/results/ranking.php?idev=58

[4] http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec.pdf

[5] http://threadingbuildingblocks.org/

[6] http://courses.csail.mit.edu/6.884/spring10/projects/lee_zhang_report.pdf

[31] J. Huang. A Case for Simple SAT Solvers. In *CP*, pages 839–846, 2007.

[32] SAT Competitions, 2011. Available at URL[7].

[33] D. Mitchell, B. Selman, and H. Levesque. Hard and Easy Distributions of SAT Problems. In *AAAI*, pages 459–465, 1992.

[34] The Caml Language, 2011. Available at URL[8].

[35] D. Syme, A. Granicz, and A. Cisternino. *Expert F# 2.0*. Apress, Berkely, CA, USA, 1st edition, 2010.

[7] `http://www.satcompetition.org`

[8] `http://caml.inria.fr/`

Appendix A

Profile of Initial Benchmark Problems

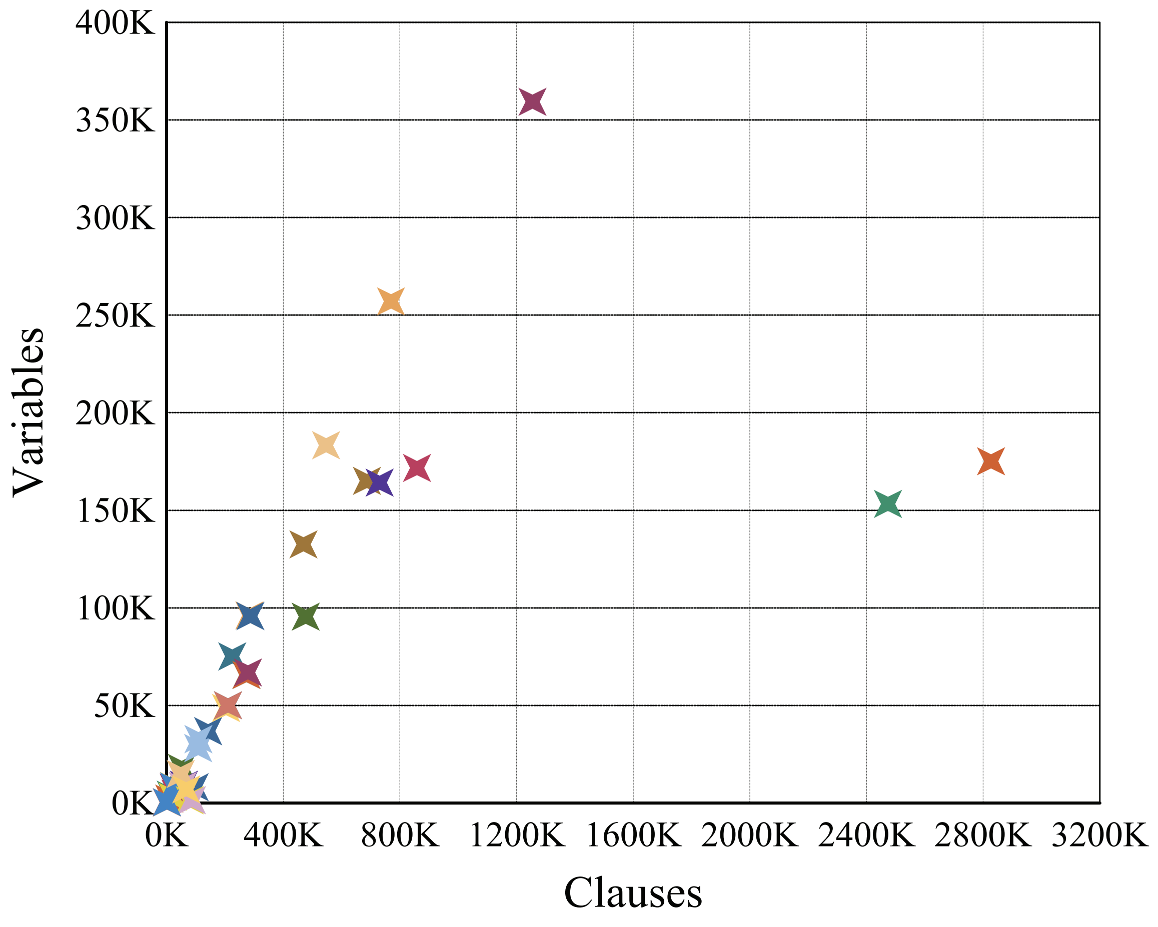

Figure A.1: Variable-Clause Distribution

			Clause Length			Clauses / Literal		
Benchmark	**Variables**	**Clauses**	**μ**	**σ**	**max**	**μ**	**σ**	**max**
aes_128_10_keyfind_1.cnf	8080	96704	5.58	1.51	8	34.20	30.65	92
aloul-chnl11-13.cnf	286	1742	2.13	1.09	11	1.53	0.53	3
am_7_7.shuffled-as.sat03-363.cnf	4264	14751	2.71	0.45	3	5.26	0.97	8
cmu-bmc-longmult15.cnf	7807	24351	2.40	0.78	18	4.25	5.84	118
countbitsarray04_32.cnf	8750	25865	2.33	0.47	3	3.91	0.82	8
goldb-heqc-alu4mul.cnf	4736	30465	3.38	1.15	16	12.09	37.96	1624
goldb-heqc-dalumul.cnf	9426	59991	3.38	1.15	32	12.47	46.00	1921
goldb-heqc-frg1mul.cnf	3230	20575	3.38	1.15	6	12.53	34.71	913
goldb-heqc-x1mul.cnf	8760	55571	3.38	1.18	70	12.13	44.25	896
hoons-vbmc-lucky7.cnf	8503	25116	2.33	0.47	3	3.29	4.39	51
hwmcc10-timeframe-expansion-k50-pdtvisns3p00-tseitin.cnf	183325	546914	2.33	0.47	3	4.06	7.58	213
ibm_fv_2004_rule_batch_30_sat_dat.k80.cnf	165064	686589	2.57	1.31	12	5.85	9.14	163
li-exam-61.shuffled-as.sat03-366.cnf	28147	108436	3.44	4.26	166	7.14	3.19	123
li-test4-100.shuffled-as.sat03-370.cnf	36809	142491	3.93	8.05	172	7.99	4.77	55
manol-pipe-c6bidw_i.cnf	96089	283993	2.33	0.47	3	4.14	5.99	329
mizh-md5-47-3.cnf	65604	273522	2.98	0.92	4	6.97	79.70	20370
mizh-md5-47-4.cnf	65604	273506	2.98	0.92	4	6.97	79.64	20356
mizh-md5-48-5.cnf	66892	279256	2.98	0.92	4	6.98	80.35	20736
mizh-sha0-35-3.cnf	48689	204067	2.99	0.91	4	7.03	68.45	15064
mizh-sha0-36-3.cnf	50073	210235	2.99	0.91	4	7.04	69.17	15438
mizh-sha0-36-4.cnf	50073	210235	2.99	0.91	4	7.04	69.12	15427
openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_1.025-notknown.cnf	95456	477186	2.38	4.13	63	5.94	13.30	161
rbcl_xits_09_unknown.cnf	1430	79453	2.97	0.21	10	52.08	94.68	379
rpoc_xits_09_unsat.cnf	1430	87044	2.97	0.20	10	52.67	96.64	390
satisfiable.cnf	360	1530	3.00	0.00	3	6.96	2.45	15
schup-l2s-abp4-1-k31.cnf	14809	48483	2.55	1.01	33	4.49	3.74	36
simon-s03-w08-15.cnf	132555	469519	2.44	1.21	15	4.90	5.59	128
slp-synthesis-aes-bottom17.cnf	32733	109177	2.55	3.43	1080	4.64	8.71	95
smtlib-qfbv-aigs-countbits128-tseitin.cnf	95810	287045	2.33	0.47	3	4.11	1.30	9
smtlib-qfbv-aigs-vs3-benchmark-s2-tseitin.cnf	257030	769313	2.33	0.47	3	4.00	4.79	258
sokoban-sequential-p145-microban-sequential.070-notknown.cnf	153284	2473656	2.28	2.97	78	8.48	13.85	108
sokoban-sequential-p145-microban-sequential.080-sat.cnf	175084	2826936	2.28	2.97	78	8.53	13.84	108
sortnet-8-ipc5-h19-sat.cnf	361125	1254773	2.43	1.99	57	4.65	26.37	1944
velev-engi-uns-1.0-4nd.cnf	7000	67586	2.83	3.15	58	11.42	35.68	728

Table A.1: Clause and Literal Distributions

Appendix B

Selected Measurements

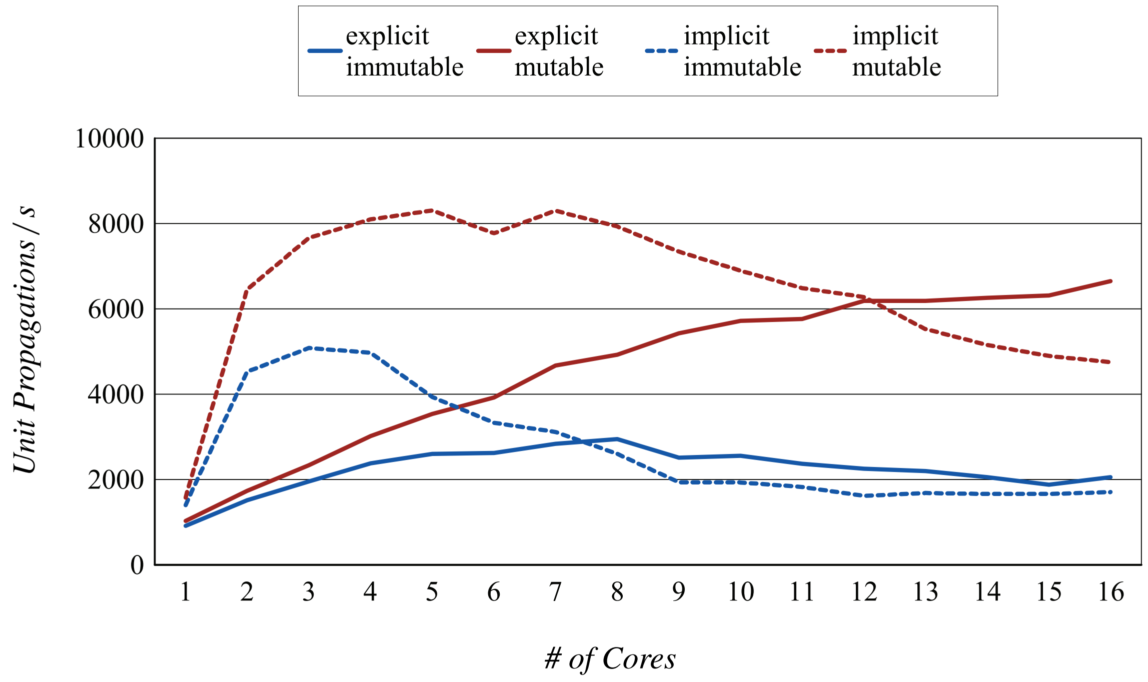

Figure B.1: Summary of 32 Bit Murinae

	Cores:	**1**	**2**	**3**	**4**
Benchmark		**6162**	**3982**	**3415**	**3228**
aes_128_10_keyfind_1.cnf	**2146**	2747	1846	1835	1857
aloul-chnl11-13.cnf	**78**	83	77	76	75
am_7_7.shuffled-as.sat03-363.cnf	**228**	296	214	189	178
cmu-bmc-longmult15.cnf	**320**	425	296	258	248
countbitsarray04_32.cnf	**331**	442	303	269	255
countbitssrl064.cnf	**2647**	3614	2314	1939	1756
dp02s02.shuffled.cnf	**68**	69	67	67	67
een-tip-sat-texas-tp-5e.cnf	**655**	854	588	495	486
goldb-heqc-alu4mul.cnf	**474**	644	430	373	364
goldb-heqc-dalumul.cnf	**890**	1219	816	687	675
goldb-heqc-frg1mul.cnf	**336**	449	305	274	259
goldb-heqc-x1mul.cnf	**822**	1127	763	638	606
hoons-vbmc-lucky7.cnf	**323**	434	295	260	248
hwmcc10-timeframe-expansion-k50-pdtvisns3p00-tseitin.cnf	**6507**	9010	5913	5140	4714
IBM_FV_2004_rule_batch_30_SAT_dat.k80.cnf	**8646**	12037	7833	6725	6292
li-exam-61.shuffled-as.sat03-366.cnf	**1669**	2234	1507	1290	1364
li-test4-100.shuffled-as.sat03-370.cnf	**2420**	3230	2192	1932	1920
manol-pipe-c6bidw_i.cnf	**3308**	4644	3027	2585	2309
mizh-md5-47-3.cnf	**3590**	5061	3254	2744	2566
mizh-md5-47-4.cnf	**3583**	5052	3249	2756	2543
mizh-md5-48-5.cnf	**3693**	5172	3357	2826	2678
mizh-sha0-35-3.cnf	**2662**	3785	2387	2032	1883
mizh-sha0-36-3.cnf	**2753**	3890	2500	2092	1963
mizh-sha0-36-4.cnf	**2748**	3894	2486	2087	1949
rbcl_xits_09_UNKNOWN.cnf	**1040**	1439	956	804	763
rpoc_xits_09_UNSAT.cnf	**1129**	1567	1027	878	826
Satisfiable.cnf	**80**	87	77	77	77
schup-l2s-abp4-1-k31.cnf	**630**	851	576	509	473
simon-s03-w08-15.cnf	**5747**	7971	5236	4529	4141
slp-synthesis-aes-bottom17.cnf	**1346**	1889	1237	1030	958
smtlib-qfbv-aigs-countbits128-tseitin.cnf	**3299**	4648	2979	2533	2360
smtlib-qfbv-aigs-VS3-benchmark-S2-tseitin.cnf	**9029**	12732	8148	6893	6491
sokoban-sequential-p145-microban-sequential.070-NOTKNOWN.cnf	**28976**	40590	26037	22246	21224
sokoban-sequential-p145-microban-sequential.080-SAT.cnf	**33184**	46781	29601	25287	24268
sortnet-8-ipc5-h19-sat.cnf	**14896**	21031	13279	11550	10658
velev-engi-uns-1.0-4nd.cnf	**887**	1231	814	689	644

Table B.1: Multi-core F# DIMACS Processing (ms)

Design	**Type**	**Cores**	**Benchmark** *Initial Size:*	**V**	**C**	**Ratio**
				Hundreds		
32 Apodemus Portfolio	immutable	**12**	aloul-chnl11-13.cnf	3	17	6.1
32 Apodemus Guiding Path	immutable	7	dp02s02.shuffled.cnf	3	7	2.1
32 Apodemus Guiding Path	immutable	**16**	satisfiable.cnf	4	15	4.3
32 Apodemus Portfolio	immutable	7	rbcl_xits_09_unknown.cnf	14	795	55.6
32 Apodemus Guiding Path	immutable	**10**	rpoc_xits_09_unsat.cnf	14	870	60.9
64 Apodemus Portfolio	immutable	4	cmu-bmc-barrel6.cnf	23	89	3.9
32 Apodemus Portfolio	immutable	4	goldb-heqc-frg1mul.cnf	32	206	6.4
32 Apodemus Guiding Path	immutable	4	am_7_7.shuffled-as.sat03-363.cnf	43	148	3.5
32 Apodemus Portfolio	immutable	4	goldb-heqc-alu4mul.cnf	47	305	6.4
32 Apodemus Guiding Path	immutable	2	velev-engi-uns-1.0-4nd.cnf	70	676	9.7
32 Apodemus Portfolio	immutable	3	cmu-bmc-longmult15.cnf	78	244	3.1
32 Apodemus Portfolio	immutable	2	aes_128_10_keyfind_1.cnf	81	967	12.0
32 Apodemus Portfolio	immutable	3	hoons-vbmc-lucky7.cnf	85	251	3.0
32 Apodemus Portfolio	immutable	3	countbitsarray04_32.cnf	88	259	3.0
32 Apodemus Portfolio	immutable	3	goldb-heqc-x1mul.cnf	88	556	6.3
32 Apodemus Portfolio	immutable	4	goldb-heqc-dalumul.cnf	94	600	6.4
32 Apodemus Portfolio	immutable	3	schup-l2s-abp4-1-k31.cnf	148	485	3.3
64 Apodemus Portfolio	immutable	4	een-tip-sat-texas-tp-5e.cnf	180	521	2.9
64 Murinae	mutable	4	li-exam-61.shuffled-as.sat03-366.cnf	281	1084	3.9
32 Apodemus Portfolio	immutable	**12**	slp-synthesis-aes-bottom17.cnf	327	1092	3.3
64 Murinae	mutable	4	li-test4-100.shuffled-as.sat03-370.cnf	368	1425	3.9
32 Apodemus Portfolio	immutable	4	mizh-sha0-35-3.cnf	487	2041	4.2
32 Apodemus Portfolio	immutable	4	mizh-sha0-36-3.cnf	501	2102	4.2
32 Apodemus Portfolio	immutable	5	mizh-sha0-36-4.cnf	501	2102	4.2
64 Apodemus Guiding Path	immutable	**16**	mizh-md5-47-4.cnf	656	2735	4.2
64 Apodemus Guiding Path	immutable	6	mizh-md5-47-3.cnf	656	2735	4.2
64 Apodemus Guiding Path	immutable	5	mizh-md5-48-5.cnf	669	2793	4.2
64 Apodemus Portfolio	immutable	3	countbitssrl064.cnf	751	2251	3.0
64 Murinae	mutable	4	openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_1.025-notknown.cnf	955	4772	5.0
64 Apodemus Guiding Path	immutable	3	smtlib-qfbv-aigs-countbits128-tseitin.cnf	958	2870	3.0
64 Murinae	mutable	4	manol-pipe-c6bidw_i.cnf	961	2840	3.0
64 Murinae	mutable	4	simon-s03-w08-15.cnf	1326	4695	3.5
64 Apodemus Portfolio	immutable	**12**	sokoban-sequential-p145-microban-sequential.070-notknown.cnf	1533	24737	16.1
64 Apodemus Portfolio	immutable	3	partial-5-11-u.cnf	1642	7307	4.5
64 Apodemus Unified	immutable	1	ibm_fv_2004_rule_batch_30_sat_dat.k80.cnf	1651	6866	4.2
64 Apodemus Unified	immutable	1	openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_1.045-notknown.cnf	1717	8588	5.0
64 Apodemus Portfolio	immutable	7	sokoban-sequential-p145-microban-sequential.080-sat.cnf	1751	28269	16.1
64 Apodemus Unified	immutable	1	hwmcc10-timeframe-expansion-k50-pdtvisns3p00-tseitin.cnf	1833	5469	3.0
64 Apodemus Portfolio	immutable	3	smtlib-qfbv-aigs-vs3-benchmark-s2-tseitin.cnf	2570	7693	3.0
64 Murinae	mutable	4	sortnet-8-ipc5-h19-sat.cnf	3593	12548	3.5

Table B.2: Best Solver - For Each Benchmark

Appendix C

Object-Oriented UML Diagrams

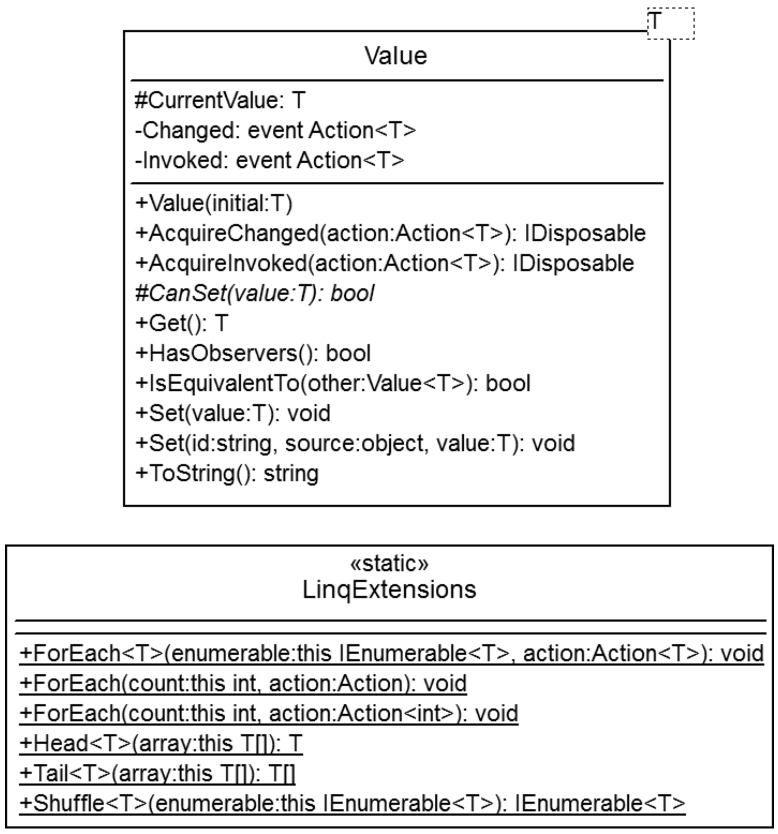
T
Value
#CurrentValue: T
-Changed: event Action<T>
-Invoked: event Action<T>
+Value(initial:T)
+AcquireChanged(action:Action<T>): IDisposable
+AcquireInvoked(action:Action<T>): IDisposable
#CanSet(value:T): bool
+Get(): T
+HasObservers(): bool
+IsEquivalentTo(other:Value<T>): bool
+Set(value:T): void
+Set(id:string, source:object, value:T): void
+ToString(): string
«static»
LinqExtensions
+ForEach<T>(enumerable:this IEnumerable<T>, action:Action<T>): void
+ForEach(count:this int, action:Action): void
+ForEach(count:this int, action:Action<int>): void
+Head<T>(array:this T[]): T
+Tail<T>(array:this T[]): T[]
+Shuffle<T>(enumerable:this IEnumerable<T>): IEnumerable<T>

Figure C.1: C# Mus Musculus - Structures (1 of 2)

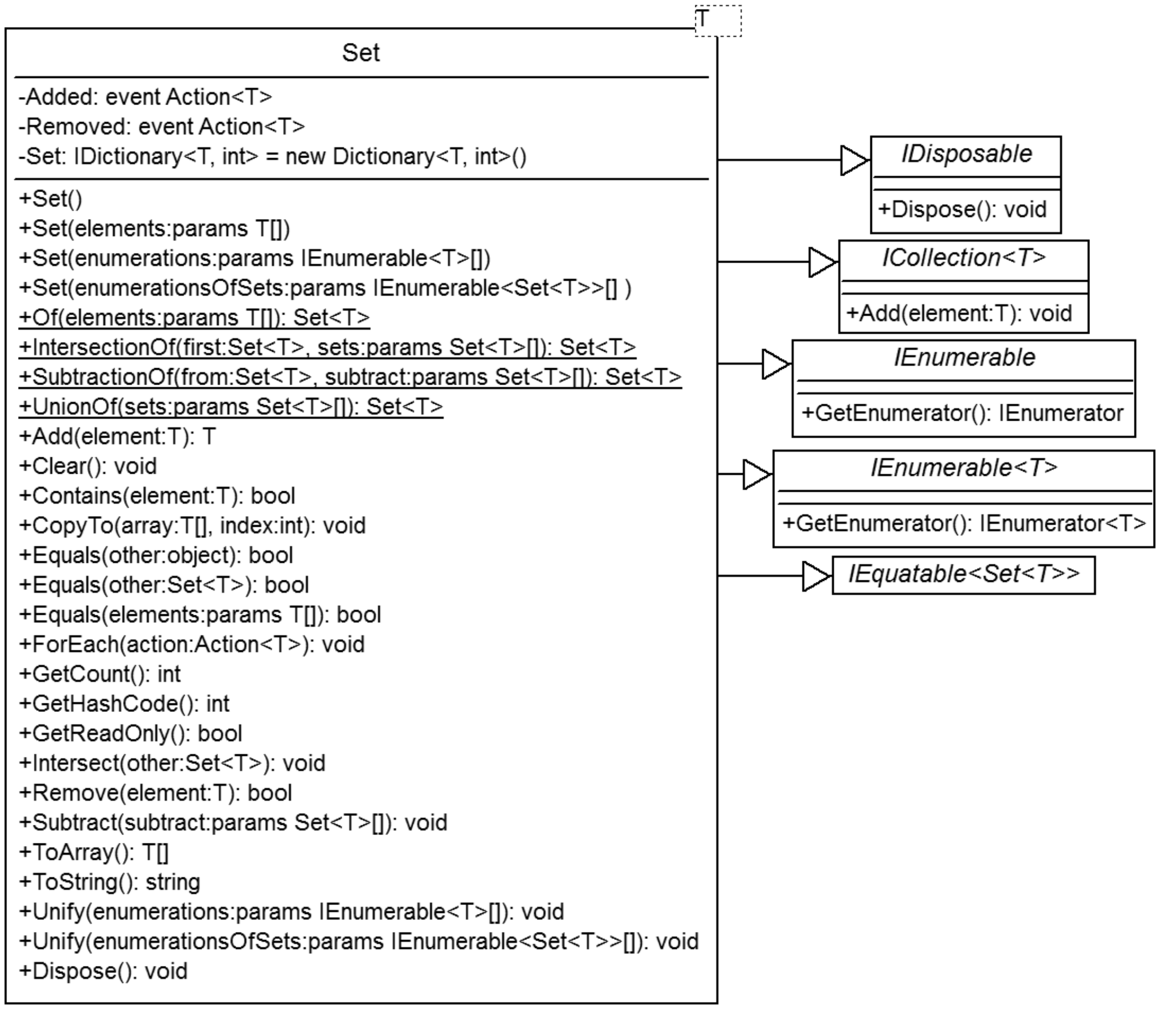
T
Set
-Added: event Action<T>
-Removed: event Action<T>
-Set: IDictionary<T, int> = new Dictionary<T, int>()
+Set()
+Set(elements:params T[])
+Set(enumerations:params IEnumerable<T>[])
+Set(enumerationsOfSets:params IEnumerable<Set<T>>[])
+Of(elements:params T[]): Set<T>
+IntersectionOf(first:Set<T>, sets:params Set<T>[]): Set<T>
+SubtractionOf(from:Set<T>, subtract:params Set<T>[]): Set<T>
+UnionOf(sets:params Set<T>[]): Set<T>
+Add(element:T): T
+Clear(): void
+Contains(element:T): bool
+CopyTo(array:T[], index:int): void
+Equals(other:object): bool
+Equals(other:Set<T>): bool
+Equals(elements:params T[]): bool
+ForEach(action:Action<T>): void
+GetCount(): int
+GetHashCode(): int
+GetReadOnly(): bool
+Intersect(other:Set<T>): void
+Remove(element:T): bool
+Subtract(subtract:params Set<T>[]): void
+ToArray(): T[]
+ToString(): string
+Unify(enumerations:params IEnumerable<T>[]): void
+Unify(enumerationsOfSets:params IEnumerable<Set<T>>[]): void
+Dispose(): void
IDisposable
+Dispose(): void
ICollection<T>
+Add(element:T): void
IEnumerable
+GetEnumerator(): IEnumerator
IEnumerable<T>
+GetEnumerator(): IEnumerator<T>
IEquatable<Set<T>>

Figure C.2: C# Mus Musculus - Structures (2 of 2)

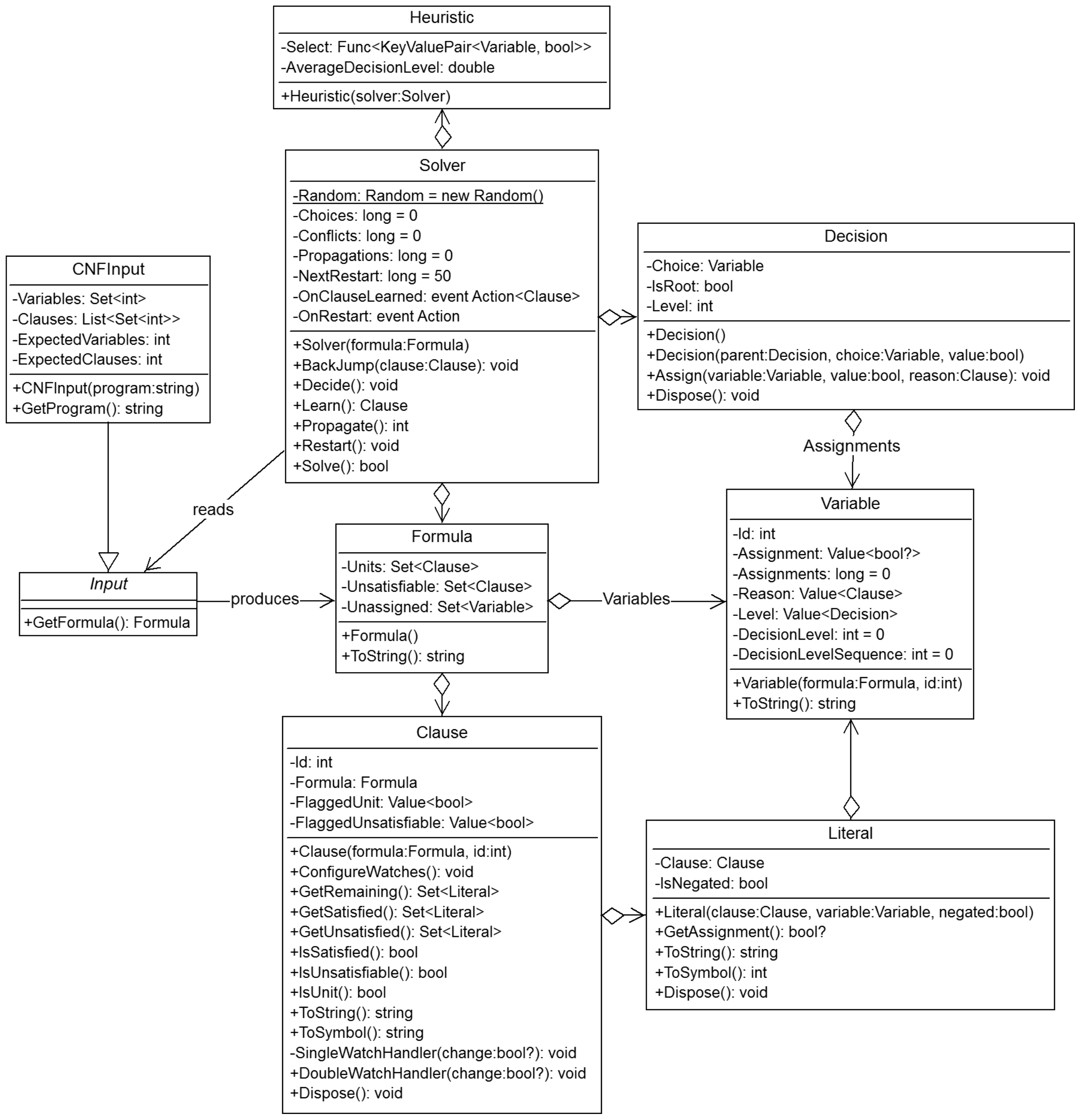

Figure C.3: C# Mus Musculus - CDCL Solver

Curriculum Vitae

Name:	Jonathan Leaver
Post-Secondary Education and Degrees:	University of Western Ontario London, ON, Canada 2009 B.Sc. Computer Science
	University of Oklahoma Norman, OK, United States 2004 B.B.A. International Business