Alef™ Verification and Planning System

Samuel B. Luckenbill, James R. Ezick, Donald D. Nguyen,
Peter Szilagyi, Richard A. Lethin
Reservoir Labs, Inc.

ABSTRACT: Recently, solvers for the Satisfiability problem (SAT) have become an enabling technology for diverse areas of military and commercial interest. However, solver performance, in terms of speed, maximum problem size, and efficiency, is a limiting factor to the more extensive application of this technology. This paper discusses Reservoir's SAT-based planning and verification system, Alef, which includes a compiler, intermediate language, and parallel solver for HPC hardware.

KEYWORDS: Alef, SAT, Satisfiability, Planning, Software Verification, Bounded Model Checking, Salt, R-Stream, Cray XD1

1 Introduction

Solvers for the Satisfiability problem (SAT) are an enabling technology for a diverse set of applications including planning, mathematics, hardware verification, and software verification. However, solver performance in terms of speed and maximum problem size limits more extensive application of the technology. We are developing a planning and verification system, Alef, which includes a compiler, intermediate language and tool, and a SAT solver for HPC hardware.

The Alef front-end compiler translates software verification problems into logical constraints. It is built on the Reservoir R-Stream™ High-Level Compiler, created for DARPA's Polymorphous Computing Architectures (PCA) project [1]. R-Stream is a C compiler, which includes an EDG front end and multiple internal representations including Static Single Assignment (SSA) [2].

To bridge the gap between front-end compilers and our SAT solver, we have developed the Satisfiability Application Logic Translator (Salt) intermediate language and tool. Salt is a macro language, similar in design to an assembly language, which provides a natural system for encoding SAT problems. Salt constraints can be translated in a single pass into Conjunctive Normal Form (CNF), preserving some high-level problem structure. Salt is applicable to multiple problem domains and provides intuitive operators for the expression of common naturally occurring constraints.

The Alef parallel SAT solver utilizes algorithms and heuristics that improve its performance over existing approaches. With assistance from Cray, Inc., we are developing the solver to run well on the Cray XD1 [3]. Our analysis shows that our algorithms combined with the low message latency of the XD1 is likely to produce a significant performance improvement over existing solvers in terms of speed and maximum problem size. In this paper, we discuss these algorithms and present our performance projections.

In Section 2, we provide motivation for the Alef system by discussing applications and social and economic need for faster SAT solvers. In Section 3, we provide some background on existing sequential and parallel SAT solvers and briefly discuss the limitations of each approach. In Section 4, we introduce the Alef parallel SAT solver and discuss its parallel algorithms. Section 5 introduces the Salt intermediate language and tool, Section 6 discusses the R-Stream based verification front end, and Section 7 covers the current status of the project.

2 Applications

The primary application domains for the Alef system are formal verification of software and automated planning. The increase in complexity of software and hardware systems is placing a greater burden on verification engineers. This trend is creating a demand for automated verification software. Additionally, designers want to verify larger sections of designs to ensure compatibility between modules and improve product quality. In the hardware verification domain, engineers are using a new class of Electronic Design Automation (EDA) tools that reduce the formal verification of complex designs to SAT problems and solve them with general-purpose SAT solvers [4]. Currently, verification costs, including empirical methods, consume in excess of 70% of the engineering effort for new hardware designs [5][6][7]. This cost is expected to grow, and verification methods are expected to shift from empirical to formal, requiring better SAT solver technology [8].

In the software domain, a major concern is the cost of undetected bugs, discovered after deployment. With the increasingly pervasive and deep embedding of computing in consumer and military devices, correctness has become a matter of public safety and national security. Expensive examples of deployed bugs include security flaws in the Windows operating system and cell phone recalls due to Java Virtual Machine bugs. In 2002, the National Institute of Standards and Technology estimated the cost of undetected bugs in software at $60 billion US dollars annually [9]. To

counter this cost, automated software verification tools are emerging, but the relative complexity of software limits its practicality.

Automated planning is a critical military technology with a direct positive impact on strategic and tactical success, mission safety, and mission cost. For example, when the Department of Defense used automated planning software to optimize the shipment of supplies and equipment during Operation Desert Storm, the savings from using computer-optimized schedules were greater than the total funding for artificial intelligence research since the 1940s [10][11]. The importance of SAT solvers for planning is illustrated by SATPLAN, the winner of the 2004 Optimal Planning Competition [12]. SATPLAN reduces planning problems specified in the Planning Domain Definition Language (PDDL) to SAT, and solves them with a general-purpose SAT solver [13].

3 Related Work

Over the past decade, a substantial amount of work has been done to accelerate SAT solvers. While the majority of this work focused on accelerating sequential solvers that run on single-processor workstations, some more recent work focuses on building hardware and software to take advantage of fine-grained and coarse-grained parallelism. In addition to the parallel algorithms presented in Section 4, the Alef parallel SAT solver includes standard heuristics from modern sequential solvers.

3.1 Sequential SAT Solvers

Most modern SAT solvers are based on the Davis-Putnam-Loveland-Logemann (DPLL) algorithm, an improved version of the original Davis-Putnam algorithm [14][15][16]. The DPLL algorithm begins with all variables in the SAT instance unassigned. It chooses a variable and assigns it either 0 or 1. After each decision, the algorithm performs Boolean Constraint Propagation (BCP). BCP is the iterative process of discovering variables that are implied to be 0 or 1 based on the current partial assignment.

Consider BCP over the following Conjunctive Normal Form (CNF) SAT instance with three clauses: $F = (\neg a \wedge b) \wedge (\neg a \wedge \neg b \wedge c) \wedge (\neg b \wedge \neg c)$. To satisfy a Boolean formula expressed in CNF, we must satisfy every clause in the formula (every clause must evaluate to 1). In order to satisfy each clause, at least one literal in the clause must be satisfied. If we assign a = 1, the first clause, $(\neg a \wedge b)$, implies that b = 1. If a = 1 and b = 1, then the second clause implies that c = 1. However, if b = 1, the third clause implies that c = 0, which is called a conflict.

In the DPLL algorithm, conflicts discovered during BCP are resolved by backtracking through the current decision and complementing its assignment. If the new assignment also leads to a conflict, the decision variable is unassigned and the previous decision is complemented. This continues until the solver finds a decision that does not lead to a conflict. If there are no previous decisions to complement, the SAT instance is unsatisfiable, and the algorithm terminates. Otherwise, the algorithm continues by choosing another unassigned variable and assigning it a value. If no unassigned variables remain, the algorithm has discovered a satisfying assignment to the variables and terminates. This algorithm is outlined in Figure 1.

```
while (true) {
    if (decide() == SAT) {
        return SAT;
    } else if (propagate() == CONFLICT) {
        if (resolve() == FAIL) {
            return UNSAT;
        }
    }
}
```

Figure 1: DPLL Algorithm

As with all NP-Complete problems, in the worst case, for a problem with n variables, the solver must search 2^n possible assignments. However, modern solvers such as zChaff and BerkMin can solve some real-world problem instances with more than a million variables [14][15][17]. This is possible because of heuristics that frequently allow the solvers to explicitly search less of the search space while still ensuring that the entire space is covered. One heuristic determines the order that the solver chooses unassigned variables and the values to assign them. zChaff and BerkMin, two industry-standard sequential SAT solvers, employ variants of the Variable State Independent Decaying Sum (VSIDS) heuristic. VSIDS tries to choose variables that are most local to the current path of the search. As a result, the solver tends to discover conflicts in the partial assignment earlier and move through unsatisfiable search space faster.

Watch lists, introduced in SATO [18] and later refined by the authors of zChaff, are a common implementation optimization to BCP for clausal SAT solvers. Watch lists allow these solvers to examine only the clauses where a conflict or implication is likely to occur. This optimization is critical because modern SAT solvers spend most of their runtime performing BCP.

Conflict-driven learning is a heuristic that occurs during conflict resolution. Normally, only a subset of the assignments to the variables lead to a conflict. By isolating that subset, the solver can prune unsatisfiable areas of the search space. Non-chronological backtracking is tightly coupled with learning. Rather than complementing the assignment to the most recent decision, the solver backtracks through enough decisions to completely resolve the conflict. The learned information

prevents the solver from re-entering the same unsatisfiable region of the search space later.

Finally, SAT solvers often use restarts to avoid becoming trapped in local minima. Because each decision in the search of a binary tree cuts the remaining search space in half, the first few decisions, when the search space is the largest, are the most important. However, the solver makes these decisions when it has learned the least amount of information about the problem. To diminish this effect, the solver may occasionally restart the search but retain the information it has learned. This gives it a chance to re-make the first few decisions with more knowledge of the search space.

3.2 Parallel SAT Solvers

PaSAT [19][20] and the parallel implementation of zChaff [21] are parallel SAT solvers designed to run on clusters. Both replicate the original problem instance on every node of the cluster and divide the search space among the nodes. The software that runs at each node is a complete SAT solver, responsible for only a subset of the variables. Each node makes decisions, performs BCP, learns, backtracks, and eventually detects that its piece of the search space is or is not satisfiable. If the search space is satisfiable, all nodes terminate and the satisfying result is reported to the user. Detecting an unsatisfiable result is more complex because the solver must show that all parts of the search space are unsatisfiable. To handle this, one processor is elected the master processor.

Because large pieces of the search space are generally unsatisfiable, these solvers incorporate load balancing strategies to keep nodes active. Additionally, they share learned information between nodes to encourage them to work synergistically. These solvers typically see speedups ranging from 1x to 20x over sequential solvers, depending on problem instance.

Inconsistent speedup is a major limitation of these parallel solvers. Even on a large cluster, these solvers are not always faster than a sequential solver running on a single workstation. This inconsistency arises because large pieces of the search space are often unsatisfiable and heuristics help solvers avoid traversing unsatisfiable search space. While the amount of space these parallel solvers can search increases linearly with added nodes, often the additional work is non-productive.

As an alternative to exploiting parallelism between multiple searches, Ying Zhao, a former graduate student at Princeton, designed an application-specific processor to take advantage of the fine-grained parallelism within a single search [22]. The Zhao architecture is a torus of Tensilica cores with distributed blocks of embedded DRAM, resembling the MIT RAW PCA architecture [23]. Each Tensilica core implements custom operators for BCP, while special routers move messages between cores. Each core acts as a BCP engine for a subset of the clauses, while a master core runs a sequential search over the entire search space. The extremely low on-chip message latency allows for the random distribution of clauses between blocks of memory without significant communication overhead during BCP. Decision making and backtracking are also parallelized. In simulations, Zhao showed speed-ups between 20x and 60x, with larger speed-ups obtained from larger problems with more available parallelism.

In all of the solvers discussed so far, the maximum size of the problem, in terms of number of clauses, is limited. While PaSAT and parallel zChaff use multiple nodes in a cluster, they must keep a complete copy of the problem instance and supporting solver state at each node. The Zhao special purpose architecture is limited to problems that fit in embedded DRAM on a chip, which is very small compared to the RAM in a workstation.

To the best of our knowledge, only one solver, part of the DiVer Bounded Model Checking (BMC) system from NEC labs, is able to use the additional memory available in parallel machines to solve problems larger than the memory of a single node [24]. DiVer's parallel solver runs a similar algorithm to the Zhao solver, where each node of a cluster is analogous to a BCP engine in the Zhao architecture. The DiVer system uses a natural linear decomposition generated from a BMC problem to distribute clauses across the nodes of the cluster. Each time step of the BMC problem is dependent only on the previous time step, such that the decomposition forms a linear chain where variables are shared only between adjacent time steps. When the clauses are distributed among cluster nodes, BCP requires only nearest-neighbor communication.

Using this approach on a network of workstations connected with Gigabit Ethernet, the authors found a performance penalty between 0.7 and 3.6. Performance penalty was measured in terms of parallel speed / sequential speed, so penalties less than 1 represent an increase in performance over a sequential solver. In their study, larger performance penalties were often related to small problem size, where less parallelism is available.

4 Alef Parallel Satisfiability Solver

Alef's parallel SAT algorithm is broken into the three threads: master, worker, and search. Each thread runs an autonomous message-driven algorithm. An instance of the Alef parallel SAT solver running on n nodes will contain one master thread, n worker threads and m search threads, where m is an integer greater than zero.

The master thread is in charge of performing preprocessing to establish the initial state of the search, distributing work to the search threads, coordinating load balancing, detecting the case when the problem is unsatisfiable, and shutting down the

solver when it is complete. Search threads are in charge of making decisions, coordinating BCP and conflict resolution, and ultimately finding a satisfying result to the problem. Worker threads perform BCP on behalf of the search and master threads, and optionally perform distributed conflict resolution. Figure 2 illustrates a possible distribution of these threads over the nodes of a Cray XD1.

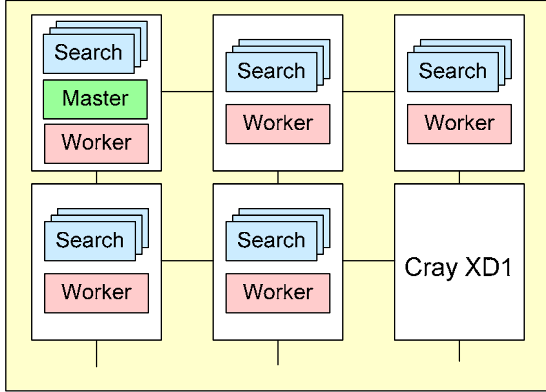

Figure 2: Example Distribution of Threads for Alef Parallel SAT Solver on HPC Hardware

4.1 Communication Library

To ensure portability and to support the message-driven model of the Alef solver, we have implemented a communication library that hides the native communication interface from the application. Because the communication latency for MPI on the Cray XD1 is exceptionally low, the library is currently implemented using MPI. The library provides one-sided send, request, respond, and receive functionality. It also supports reclamation of storage from outgoing messages, pooling of MPI handles, a priority queue for each application-level thread, invalidation of responses to stale requests, and termination detection.

One-sided communication is critical to the performance of our application because its behavior is data-dependent. For example, during BCP, messages are sent from the search thread to various worker threads depending on the distribution of variables between nodes. Messages are then forwarded between worker threads as additional implications are discovered. To support this behavior efficiently, each thread must be able to send a message and continue on with other work. On the receiving side, a thread must be able to receive a message when it becomes idle. While MPI supports non-blocking sends and receives, it is a two-sided communication protocol and there is no consideration for memory management of pending sends. For each send, the recipient must actively receive the message before the sender can free the message state and MPI handle. Our library provides pools of memory-managed handles for sending messages and maintains queues of incoming messages, making the details of the communication protocol transparent to the application.

The API additionally defines two other functions, *request* and *respond*. These functions enable a thread to send a single request that may provoke multiple responses. The thread may, at any time, query the communication library to determine if all responses to a request have been received. Because the algorithm is message driven, a completed request indicates that all work from that request has terminated. We call this feature termination detection, and its application to distributed BCP is described in Section 4.2. Finally, because the communication library keeps track of requests with pending responses, it is possible to invalidate responses to a stale request. As stale responses arrive, they are not queued and the application does not receive them.

4.2 Distributed Boolean Constraint Propagation

DPLL-based SAT solvers spend upwards of 80% of their time performing BCP [14]. In a modern sequential SAT solver, BCP is performed after each decision is made. The solver starts by fetching clauses in the watch list for the variable that was assigned. These are the clauses that could result in an implication or a conflict. Because BCP is iterative, each decision may lead to multiple implications, which may lead to more implications, and so on.

Similar to the DiVer solver, Alef's BCP algorithm is distributed and utilizes multiple BCP engines in parallel. Each engine, which we call a worker thread, owns a partition of the clauses from the SAT instance and is responsible for performing BCP on only those clauses. Unlike DiVer, Alef has multiple search threads, so worker threads must maintain the state necessary to perform BCP for each search thread. This state includes watch lists, assignments to variables, and partial assignment stacks. The overhead of this state is not excessive; each small watch list is approximately $1/n$th the size of a full watch list, where n is the number of search threads. If a worker thread is able to do BCP for all n search threads, the total storage requirement is equivalent to one full watch list.

In Figure 3, we have distributed the clauses from our sequential BCP example in Section 3.1 between two worker threads. Worker 1 owns clauses $(\neg a \wedge b)$ and $(\neg a \wedge \neg b \wedge c)$, and Worker 2 owns the clause $(\neg b \wedge \neg c)$. Distributed BCP begins when the search thread sets a = 1, then sends a BCP request message to the worker threads that contain the variable a. Because only Worker 1 contains the variable a, it sends only one message. Worker 1 responds by performing BCP on the two clauses in its clause database, which lead to the implications $b = 1$ and $c = 1$. Rather than sending the new implications back to the search thread, Worker 1 checks to see if any other worker threads have variables b or c in their clause partitions. Because Worker 2 has both b and c, the

message with the entire chain of implications is forwarded to Worker 2. When Worker 2 sets b = 1, its clause implies that c = 0, which conflicts with the implication c = 1 from Worker 1. Upon discovery of the conflict, BCP halts and the conflict is sent back to the search thread.

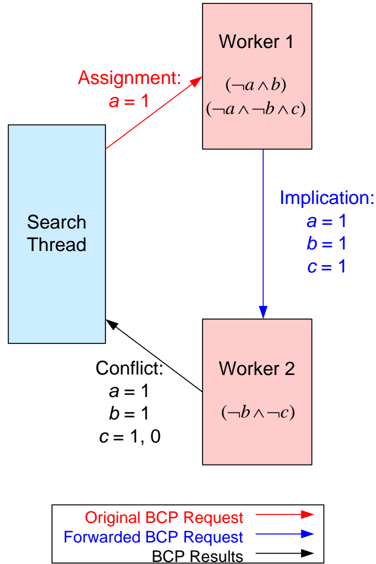

Figure 3: Distributed Boolean Constraint Propagation

Implication forwarding makes it difficult for the search thread to determine when BCP has terminated. For one request sent, the search thread may receive multiple responses. In Alef, we handle termination detection by assigning each message a fractional value of the total response. As responses to a BCP request are received, the fractional values are summed. When the sum reaches 1, the search thread knows that all responses to its BCP request have been received.

In Figure 4, a search thread issues a BCP request to a single node. The message starts with a fractional value of (1/1), indicating that it represents the entire request. The implied variable(s) discovered by Worker 1 exist in the partitions of both Worker 2 and Worker 3. Before forwarding the implications to Worker 2 and Worker 3, the fractional value is divided by 2, such that each forwarded message has a value of ½. Similarly, Worker 2 discovers another implied variable contained in the partition of two other worker threads. The fraction is split to ¼, and the implications are forwarded back to Worker 1 and on to Worker 4. When Worker 3 receives the forwarded message from Worker 1, the implied variable(s) it discovers only exist in the partition of Worker 2. The fraction is not split and the message is forwarded on. When each chain of BCP terminates, the entire list of implications is sent back to the search thread. The fractional values are summed after each message arrives, and the thread continues with its search after the values sum to 1. Fractional termination detection is handled entirely within the communication library.

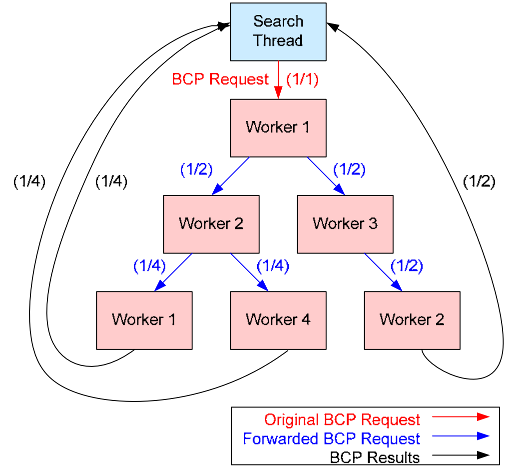

Figure 4: Message Forwarding and Termination Detection

4.3 Conflict Resolution

One of the most important heuristics in modern SAT solvers is their ability to learn. The sequential implementation of conflict resolution in zChaff assumes that clauses are stored locally along with a complete assignment stack [14]. The assignment stack is a data structure that records the order in which decisions are made and implications are discovered, and the clause that resulted in each implication. Like the DiVer parallel solver, Alef currently implements a sequential conflict resolution algorithm at each search thread.

Though DiVer's solver is parallel with distributed data, it implements a sequential conflict resolution algorithm. This requires copying clauses from the nodes where they are stored to the master processor in charge of the search. Additionally, the master processor must construct an assignment stack from implications that are discovered in parallel at worker nodes. In the DiVer solver, as BCP engines discover implications, both the implications and the clauses that resulted in implications are returned to the master thread along with enough ordering information to construct the assignment stack.

Alef has a more complex BCP algorithm than DiVer's solver, making construction of a global assignment stack more difficult. Rather than sending implications back to the search thread as they are discovered, they are forwarded to other nodes until the chain of BCP terminates. As a result, messages received in response to a single decision may contain duplicate implications. Additionally, we do not assume that any channels deliver messages in order, allowing us to use lower-level communication libraries such as active messages.

To build the assignment stack, we must construct a topological sort of the implications such that no implication discovered as a result of another implication appears before it in the stack. In

addition to the assignment stack, Alef needs some method to fetch the clauses necessary for conflict resolution. The current implementation requests clauses one-by-one, as they are needed. This has the benefit of requiring no significant amount of extra storage on the search thread's node. However, we plan to experiment with aggressive forwarding schemes similar to the DiVer approach for better performance.

While the sequential conflict resolution algorithm is sufficient for correctness, it does not take advantage of the distributed data or parallel hardware available in the Alef system. To address this, we have developed a distributed conflict resolution algorithm. This algorithm does not copy clauses to the search thread, nor does it construct a global assignment stack. While a complete description of the algorithm is beyond the scope of this paper, it is similar to a distributed min-cut max-flow algorithm [25].

4.4 Alef and FPGAs

The Cray XD1 features Field-Programmable Gate Arrays (FPGAs) connected to Opteron processors through Hypertransport [3]. We have developed several ideas for the use of FPGAs in conjunction with a general-purpose processor for our SAT solver, though limitations in the software and hardware that support FPGAs prohibit us from implementing these ideas in the immediate future.

One promising idea is the fast evaluation of unrolled circuits for hardware BMC. In hardware BMC, we want to verify that a circuit conforms to a specification, for example, that an error signal is always 0. Because proving that the error signal will always be zero is undecidable, we bound the problem to k time steps from some initial state. In a typical SAT-based BMC tool, the circuit is unrolled k times and translated to CNF. The tool then adds clauses to assert that the error signal is always 1. The entire problem instance is passed to a SAT solver, which tries to find a satisfying assignment. If the solver is able to find a satisfying assignment, it has found a case where an error could occur. An unsatisfiable result from the solver proves that there can be no error within k time steps from the initial state.

One area where FPGAs excel and general-purpose processors are weak is in the propagation of values through Boolean circuits. In a general-purpose processor, we have to build software models of the behavior of each gate and a data structure that represents the connections between the gates. To propagate a signal through this "circuit," we traverse the data structure of connections evaluating the appropriate gate models along the way. This can be made quite fast by compiling the circuit rather than interpreting it, as is done in high-end Verilog simulators. However, it is still significantly slower than a FPGA where the models of gates are replaced by actual gates and propagation is massively parallel.

We suggest that each unrolling of the circuit could be compiled to a FPGA for fast propagation. Each unrolling is a copy of the circuit with the outputs from the previous copy connected to the inputs of the next. To use a FPGA, we would only have to synthesize one copy, then place and route it once for each time step. The general-purpose processor running the SAT solver could request that an attached FPGA propagate signals through the circuit and return the result of the propagation. However, in order to allow for propagation of partial signals, the circuit would have to be translated into ternary logic (X/0/1) before compilation to the FPGA. This translation would be inexpensive, but require about twice as many gates on the FPGA. Additionally, SAT solvers propagate signals backwards through circuits starting at arbitrary points. For this, we would need a new circuit for each input to the original circuit. While building these circuits would not be particularly computationally expensive, they require significant real estate on the FPGA.

In addition to the requirement for ternary logic and reverse propagation, the time it takes to do the place and route and the latency between the general-purpose processor and the FPGA represent significant obstacles. For a tightly packed large Xilinx FPGA, place and route can take multiple hours. Because we must count this overhead in the total runtime of our solver, the costs could easily outweigh the benefits of using a FPGA, especially for smaller problems.

On the XD1, the latency for a round trip message from the Opteron to the FPGA is approximately 0.66μs, which is 3000 clock cycles on a 4GHz general-purpose processor. In addition to the 3000-cycle overhead, the FPGA still has to do the propagation. Because of the overhead, it may have been faster to do the propagation directly on the Opteron. While we cannot perform a cost-benefit analysis for using the FPGA without significant additional effort, the incremental overhead for its use due to chip-to-chip latency in addition to the up-front overhead due to place and route appear excessive.

In the future, we believe that compilation of the entire DPLL-based SAT solver to the FPGA may be a promising path. SAT solvers are relatively simple, consisting of a state machine, propagation engines, and heuristics for making decisions and resolving conflicts. Despite their algorithmic simplicity, their implementation often requires many thousands of lines of code in a high-level programming language. Compilation of the solver to a FPGA would be best accomplished by a compiler capable of translating a higher-level programming language into gates.

Another technology that could enable SAT on FPGAs is a fast dynamic compiler combined with support for quickly reprogramming small areas of the FPGA. Using this tool, we could compile small pieces of data and program them on the FPGA as they become important to the search. However,

effective use of this compiled data would require significantly lower processor-to-FPGA latency.

4.5 Projected Performance

We believe the Alef solver will be able to handle larger problems and solve problems faster than sequential SAT solvers. A workstation running a sequential SAT solver is limited to problems that fit into memory and to the use of a single CPU. The Alef system is able to use all available memory and CPUs on a supercomputer or in a cluster. In terms of problem size, we should be able to attempt to solve problems that are up to two orders of magnitude larger than other solvers, assuming more than 100x available RAM on a large HPC system compared to a workstation.

In terms of speed, we expect Alef will excel on large problems, where it is possible to exploit both data parallelism in BCP and the algorithmic parallelism of multiple synergistic searches. Quantifying the performance advantage of our solver is difficult, and ultimately we must measure its performance on benchmarks and real-world applications. However, we can estimate the solver's performance by comparing existing solvers that share similar design characteristics.

The parallel solver from NEC Lab's DiVer system exploits data parallelism in BCP using a cluster of workstations. The system runs a single search, controlled by a master processor, which controls multiple BCP threads, one per cluster node. The system was benchmarked using a cluster of three Pentium 4 workstations with standard Gigabit Ethernet, which has an MPI message latency of approximately 100µs.

Ganai and Gupta report a performance penalty of 0.7 to 3.7 compared to a sequential solver over a range of benchmarks. Performance penalty was defined as parallel speed / sequential speed, so penalties less than 1 represented a speedup. For all of the benchmarks under 124MB—benchmarks that would fit on a single node with a sequential solver—they found a significant slowdown (penalty >= 1.4). The largest slowdown occurred on a very small 8MB benchmark, where the overhead of distribution versus solver runtime was high. They saw consistent speedup on benchmarks over 254MB. One outlying 1538MB benchmark showed a slowdown (penalty = 1.4).

For large problems, DiVer's solver often produced a speedup. In the worst case, for a very small benchmark, it produced a moderate slowdown (40%). This is encouraging because the MPI message latency of standard Linux with Gigabit Ethernet is over 50x greater than the latency of the Cray XD1 (1.7µs) [3]. Because of this, we believe that targeting Alef to the XD1 will allow the solver to consistently realize a performance speedup from data parallelism in BCP on moderate to large structured benchmarks.

Calculating how much speedup is difficult, but we can set an upper bound by looking at the Zhao architecture from Princeton University. Zhao's application-specific architecture is a torus of customized Tensilica cores, each of which has about the same power to run BCP as a workstation. The chip runs a similar algorithm to the DiVer solver, but unlike DiVer (and Alef), Zhao did not have to partition the clauses because the on-chip latency between processor cores is in the nanosecond range. The extremely low communication overhead allowed Zhao to exploit all available data parallelism in BCP. Using Tensilica's simulators, Zhao showed the maximum speedup from parallelism in BCP was 30x - 60x over a range of benchmarks.

From Zhao's work, we can conclude that, in the best case, with very low communication latency, we can expect 30x - 60x speedup due to data parallelism from BCP. From Ganai and Gupta's work, we know that the worst case is a 40% slowdown. We expect the Alef solver on the Cray XD1, with a latency between the two bounding examples, will see a speedup between 1x and 30x from data parallelism alone.

The second form of parallelism that our system will exploit is algorithmic. By dividing the search space between multiple search threads, each thread can search for a solution to the same problem in parallel. This type of solver relies on various forms of load balancing and sharing of learned information, and most examples get between 1x and 20x speed up over sequential solvers. Because the Alef solver is multithreaded and will employ similar load balancing and information sharing algorithms, we expect 1x - 20x speedup for this type of parallelism.

While these two types of parallelism are independent, we do not expect the speedups to multiply in the Alef solver. This is because the solver has a finite number of worker threads. Each added search thread places a greater load on the worker threads, which limits the amount of parallel BCP they can perform on behalf of each search thread. To achieve the maximum 30x – 60x speed up from data parallelism in BCP, we would need 30 – 60 BCP threads working together on behalf of one search thread, each requiring a processing node. However, we expect message latency to limit parallelism in BCP, and expect additional search threads will help hide this latency.

5 Salt Language and Tool

Salt is a constraint logic language and translator for SAT applications. The Salt input language provides a simple and intuitive syntax for representing propositional logic, fixed point arithmetic, and set theoretic constraints. The Salt translator provides a uniform method of translating those constraints into optimized CNF. Salt can also introduce partition annotations making it a useful preprocessor for the Alef parallel SAT solver. Because the Salt language has an

application-independent syntax, we can extend the functionality of the Alef parallel solver without modification to application problems expressed in Salt. Instead, we need only modify the Salt translator to exploit the new solver functionality.

Shaker, a companion tool to Salt, parses the output of a SAT solver and translates the resulting variable assignments into a readable form according to the variable bindings provided in a Salt input file.

Now at version 1.02, Salt optimizes constraints based on a system of lazy inference. In this system, assignments that can be inferred from already-asserted constraints are used to simplify subsequent constraints. The resulting translator is both efficient and powerful. Salt also provides a novel optimization mechanism based on controlling the complexity of logical constraint subexpressions. This parameter, the capture threshold of the translation, is key to achieving an optimal translation for a specific solver.

The next version of Salt will provide a more robust intermediate representation that can be generated directly though an API, offering the ability to bypass the Salt language front-end. The IR will also support a more aggressive set of constraint optimizations, targeted at numerical and pseudo-Boolean (linear inequality) constraints.

6 Alef Compiler

SAT solvers have applications in a wide variety of problem domains, including planning and formal verification. For Alef, we have chosen three domains that we think are particularly interesting for their military and commercial applicability: planning, software bounded model checking, and formal verification of numerical precision.

The Alef system will include a front end that will accept PDDL and translate it to CNF, annotating it for distribution over the nodes of a parallel machine running our SAT solver. Optimal planning problems have a natural time-step structure, where each time step is only dependent on the previous time step. With permission of the author, we have modified SATPLAN [13] to annotate the CNF it produces with time step markers. SATPLAN accepts optimal planning problems in PDDL, produces CNF, then calls a sequential SAT solver to solve the SAT instance. We believe that our parallel solver will be able to solve the annotated CNF files from SATPLAN significantly faster than a sequential solver is able to solve un-annotated CNF.

For software BMC, we are modifying Reservoir's R-Stream™ optimizing compiler to do automatic model extraction from the source program and its specification. From these two elements, R-Stream generates a Boolean formula that is unsatisfiable if the specification holds for all possible executions of the program. For the extraction, we use R-Stream's SSA internal representation, which simplifies the extraction process. R-Stream also allows us to apply standard compiler transformations such as program slicing, constant propagation and variable hoisting to increase the performance of the underlying model checker.

To improve the scalability of our system, our system will include automatic program slicing to remove irrelevant information from the generated problem. Additionally, we will implement a program encoding that preserves context-sensitivity by explicitly modeling the program stack. This will allow us to skip inlining of procedure calls.

Formal verification of numerical precision builds on software BMC techniques, particularly the ability to model the exact value of a variable at any given point in a bounded program. Our technique generates two value-exact models of the program, one with lower precision and one with higher precision. The high-precision version is identical to the low-precision version except that bit-width of numeric types have been increased. We then use our software bounded model checker to verify that the result of the low-precision version does not differ from the high-precision version by more than an error bound e. If this property holds, we have proven that the low-precision program approximates the high-precision program within that error bound.

Extracting value-exact models of a program is only possible if the execution of the program can be statically bound. For most programs in which numerical precision is important, this will not be an issue. A typical application might be building a hardware circuit to implement a software function. Because it is possible to implement the function in hardware, it should be possible for the compiler to statically bind the execution of the function, through programmer pragmas or static analysis.

7 Project Status

As of May 2006, 13 months of our 23-month Phase II SBIR contract have elapsed. A sequential version of the Alef SAT solver is complete and the parallel version is nearing its alpha release. The Salt and Shaker tools are now at version 1.02 and we are currently working towards 2.0. SATPLAN has been modified to support our parallel solver, and we expect to demonstrate the verification front end in the coming months. We are actively soliciting prospective clients and collaborators.

About the Authors

Samuel Luckenbill is building the Alef Parallel SAT solver and is the project manager for Reservoir's DARPA Phase II SBIR to build the Alef System. He can be reached at sbl@reservoir.com.

James Ezick, Ph.D., works on Salt and other formal verification projects at Reservoir. He can be reached at ezick@reservoir.com.

Donald Nguyen and Peter Szilagyi are using Reservoir's R-Stream compiler to verify program correctness via bounded model checking with SAT. They can be reached at nguyen@reservoir.com and szilagyi@reservoir.com.

Richard Lethin, Ph.D., President of Reservoir Labs, is the primary investigator for the DARPA Phase II SBIR. He can be reached at lethin@reservoir.com.

For voice, fax, or written inquiries, all authors can be reached at: Reservoir Labs, Inc., 632 Broadway, Suite 803, New York, NY, 10012, +1-212-780-0527 (voice), +1-212-780-0547 (fax). http://www.reservoir.com.

References

[1] R. Lethin et al., "R-Stream 3.0: Technologies for High Level Embedded Application Mapping," Reservoir Labs, Inc., New York, NY, May 2004.

[2] C. Click, M. Paleczny, "A Simple Graph-Based Intermediate Representation," The First ACM SIGPLAN Workshop on Intermediate Representations, San Francisco, CA, 1995.

[3] Cray, Inc, "Cray XD1 Datasheet," 2005, Available at: http://www.cray.com/products/xd1/index.html

[4] R. Goering, "Synopsys Tips 'Hybrid' Formal Verification," EE Times, May 2003.

[5] T. Anderson, R. Bhagat, "Tackling Functional Verification for Virtual Components," ISD Magazine, Nov. 2000.

[6] P. Rashinkar, L. Singh, "New SoC Verification Techniques," In Proc. IP/SOC'01, Mar. 2001.

[7] F. Ozguner, D. Marhefka, J. DeGroat, B. Wile, J. Stofer, L. Hanrahan, "Teaching Future Verification Engineers: The Forgotten Side of Logic Design," In Proc. DAC'01, Jun. 2001.

[8] T. Schubert, "High Level Formal Verification of Next-Generation Microprocessors," In Proc. DAC'03, Jun. 2003.

[9] National Institute of Standards and Technology. "The Economic Impacts of Inadequate Infrastructure for Software Testing," May 2002, Available at: http://www.nist.gov/director/prog-ofc/report02-3.pdf

[10] E. Phillips, "If it Works, it's Not AI: A Commercial Look at Artificial Intelligence Startups," M.S. Thesis, MIT, 1999.

[11] P. Winston, Artificial Intelligence, [In Lecture], MIT, 1992.

[12] "International Planning Competition," [Online Document], Hosted at ICAPS'04, Jun. 2004, Available at: http://ipc.icaps-conference.org/

[13] H. Kautz, B. Selman, "Unifying SAT-based and Graph-based Planning," In Proc. IJCAI'99, Aug. 1999.

[14] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: Engineering an Efficient SAT Solver. In Proc. DAC'01, Las Vegas, NV, Jun., 2001.

[15] E. Goldberg, Y. Novikov, "BerkMin: a Fast and Robust Sat-Solver," In Proc. DATE'02, Mar. 2002.

[16] L. Ryan, "Efficient Algorithms for Clause-Learning SAT Solvers," M.Sc. Thesis, School of Computing Science, Simon Fraser University, Feb., 2004.

[17] "Boolean Satisfiability Research Group at Princeton," [Online Document], Apr 2006, Available at: http://www.princeton.edu/~chaff/zchaff.html

[18] H. Zhang, "SATO: An Efficient Propositional Prover," In Proc. CADE'97, Townsville, Australia, Jul. 1997.

[19] C. Sinz, W. Blochinger, W. Kuchlin, "PaSAT – Parallel SAT-Checking with Lemma Exchange: Implementation and Applications," Vol. 9 of Electronic Notes in Discrete Mathematics, Boston, MA, Jun. 2001.

[20] W. Blochinger, C. Sinz, W. Kuchlin, "Parallel Propositional Satisfiability Checking with Distributed Dynamic Learning," Parallel Computing 29, 2003.

[21] Y. Yu, L. Zhang, S. Malik, "Practical Experiences with Parallel Boolean Satisfiability by Shared Learning," SRC Report, Pub P006114, Aug. 2003.

[22] Y. Zhao, "Accelerating Boolean Satisfiability through Application Specific Processing," Ph.D. Thesis, Department of Electrical Engineering, Princeton University, Oct. 2001.

[23] M. Taylor, et al. "The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs," IEEE Micro Mar./Apr. 2002.

[24] M. Ganai, A. Gupta, Z. Yang, P. Ashar, "Efficient Distributed SAT and SAT-based Distributed Bounded Model Checking," In Proc. CHARME'03, Oct. 2003.

[25] R. Ahuja, J. Orlin, "A Fast and Simple Algorithm for the Maximum Flow Problem," Operations Research, Vol. 37, No 5, Sept. – Oct. 1989.

This work was produced with US government support, under DARPA Contract W31P4Q-04-C-R257. The US government has certain rights to this work.