Abstract

This paper presents DAHL, a programming environment for the development of distributed systems. DAHL provides a high-level language that is specialized for the implementation of distributed systems, and a stable and high-performance runtime that can efficiently execute DAHL programs in a production environment.

The DAHL language is declarative, event-triggered and rule based, which enables concise, albeit clear, implementations of distributed applications. The DAHL language features the usual control-flow statements, expressive data types, as well as a set of network and security related built-in predicates.

State-of-the-art languages to implement distributed systems include Mace, which is based on C++ and thus lacks aptitude for formal verification, and P2, which is based on Datalog, a query language that is not expressive enough to implement interesting applications. Moreover, the current P2 implementation is not efficient enough for non-purely network bound applications.

DAHL combines the performance and expressiveness of Mace with the succinctness of P2 to create a better and practical programming environment for the development of distributed applications.

To demonstrate the power of DAHL, we implemented several applications of different representative categories. This paper presents two of them: Chord, a distributed hash table, which is a network driven protocol, and D’ARMC, a distributed software model checker that is CPU-bound.

1 Introduction

Building a high-performance, fault-tolerant distributed application is a difficult and error-prone task (see, e.g., [18, 27]). The programmer has a lot of implementation details to consider, such as memory management, socket programming, message serialization, and operating systems subtleties, which are unrelated to the protocol specification itself. Moreover, a distributed application has to deal with ever-changing network conditions, node and link failures, asynchrony, and possible malicious behavior from other nodes in the system.

Chandra et al. [3] recently published their experience in building a real-world high-performance database using the Paxos protocol [11]. They note that, even with the authors’ experience in building complex distributed systems and that the Paxos protocol was proposed and proven correct more than 15 years ago, implementing such a high-performance and robust database turned out to be a non-trivial multi-year effort. They also note that the main reason behind the complexity was the lack of appropriate tools that can help practitioners correctly implement these protocols.

Software tools can make the development of distributed systems an easier, less error-prone, and more productive activity, while improving the quality of the produced code. High-level programming languages supported by analysis and verification tools hold vast potential towards achieving these goals (e.g., [10, 13, 16, 26, 29]). Unfortunately, the existing approaches do not provide a comprehensive solution to the challenge by sacrificing some of the necessary attributes, like language expressiveness, succinctness of implementation, efficiency, or aptitude to correctness checking.

In this paper, I present DAHL, a programming system specialized for the development of distributed applications. It pursues three design objectives: make distributed systems easy to write, easy to understand and analyze, as well as easy to execute efficiently. To achieve these goals, DAHL
offers a high-level programming language providing constructs for writing distributed applications, and a runtime environment for efficient program execution.

The DAHL language is based on Prolog, a declarative logic language. Declarative languages often result in simpler, clearer and more compact programs [12]. DAHL extends Prolog with specific predicates and other language constructs to enable efficient construction of networked programs. Moreover, DAHL extends Prolog with network-driven queries, opposed to the traditional user-driven queries.

The programming model of DAHL is event-driven, i.e., a program can send and receive events through event handlers. We believe this is a good model for the development of distributed programs, as it is similar to how protocols are usually specified (diagrams of messages that are exchanged between nodes), and because it holds potential for code reuse between programs, which is usually not possible or not efficient with other programming models (e.g., finite state machines).

By layering on top of an existing language, we can reuse its existing tools and runtime. In particular, the DAHL runtime is based on SICStus Prolog [24], which is an industry standard compiler. We use its compiler as well as its Prolog runtime environment (the blackboard, the facts database, the garbage collector, etc.). The networking backend is based on libevent [15], which natively delivers application events when a network message arrives or when a timer expires. By being event-based and by reusing industrial strength tools, DAHL can achieve high throughput, scalability, and reliability, as it is demonstrated by the tests and benchmarks that we run.

In order to show that faithful implementations of complex distributed applications can be quickly built with DAHL, and that DAHL is expressive enough for a wide range of applications, we have implemented several distributed applications. In this paper I present two of those: Chord, a peer-to-peer (P2P) distributed hash table (DHT) that is a network-driven protocol, and D’ARMC, a distributed software model checker that is CPU-bound. These applications were developed in a few days and their code size is at least an order of magnitude smaller than other implementations in more traditional languages, such as C. Moreover, these applications perform well when compared with the reference implementations. In case of Chord, the latency of lookups and the consistency levels are similar with the reference implementation, while D’ARMC achieves good speedups and efficiency levels. To our better knowledge, D’ARMC is the world’s first scalable software model checker.

2 Related Work

Over the last few years, several specialized languages for the implementation of distributed systems have been proposed. Often called domain-specific languages (DSL), the list includes, for example and in no special order, Linda [2], nesC [6], MACEDON [20] and Mace [10], Acute [21], and P2 [13,14].

These languages are very different both in syntax and semantics. So, we identified several key aspects to better compare them. Table 1 summarizes our findings.

3 The DAHL Language

In this section, a comprehensive description of the DAHL programming language is given. The DAHL language is based on Prolog [22,24].

The presentation is divided in three parts. First we introduce the language with an example. We then describe how data is represented and manipulated by programs, and finally we present the syntax and semantics of DAHL.

3.1 DAHL by Example

As an introductory example to the DAHL language, we present an implementation of the Bully protocol [5].

A quick introduction to the protocol is given following Figure 1. In step 1, node ‘a’ detects that node ‘c’ (the current leader) failed. Therefore it broadcasts an ‘election’ message to all its neighbors with an identifier greater than its own (in this case, nodes ‘b’ and ‘c’). Node ‘b’ receives the message from node ‘a’ and replies with an ‘OK’ message (step 2). At the same time, node ‘b’ starts the election protocol in the same way that node ‘a’ did in step 1 (step 3). Finally, as node ‘b’ does not receive any reply during the timeout period, it becomes the leader, and broadcasts itself as such (step 4).
Table 1: Feature comparison between Linda, nesC, MACEDON, Mace, Acute, P2, and DAHL.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Linda</th>
<th>nesC</th>
<th>MACEDON</th>
<th>Mace</th>
<th>Acute</th>
<th>P2 (pure)</th>
<th>DAHL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Language</td>
<td>Prolog</td>
<td>C++</td>
<td>C++</td>
<td>C++</td>
<td>OCamel</td>
<td>Datalog</td>
<td>Prolog</td>
</tr>
<tr>
<td>Expressiveness</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Succinctness</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Bug finding</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Property proving</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Efficiency</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Prog. Model</td>
<td>any</td>
<td>events</td>
<td>FSM</td>
<td>FSM</td>
<td>any</td>
<td>events</td>
<td>no</td>
</tr>
<tr>
<td>Concurrency</td>
<td>no</td>
<td>pseudo</td>
<td>lock based</td>
<td>between layers</td>
<td>no</td>
<td>events</td>
<td>no</td>
</tr>
</tbody>
</table>

Figure 1: Sample execution of the Bully protocol.

Figure 2 shows the full source code of the implementation of the protocol in DAHL. Line 1 declares the node table, which consists of pairs in the form (Id, Address). To simplify the example, we assume that each node has its own table populated with information about all the nodes in the system.

Lines 4–9 implement the election protocol: lines 6–8 broadcast an ‘election’ message to all nodes with an identifier greater than its own, and line 8 asks the system to execute the timeout predicate if no ‘ok’ message arrives within 3 seconds. The election predicate is the one that should be called whenever a node detects that the leader crashed (not shown in this example).

Lines 11–13 define the event handler for the ‘election’ event. It sends an ‘ok’ message back to the sender of the event (line 12), and then runs the election protocol (line 13).

Lines 15–16 define the event handler for the ‘elected’ event. It simply logs that there is a new leader. In a real system, this handler would probably also need to save the new leader to some variable.

Finally, lines 18–23 define the predicate that is called when no ‘ok’ message was received within the timeout period. Line 19 queries the system for the node’s address, and line 20 retrieves the node’s identifier from the table out of the address. Line 21 logs the event, while lines 22–23 broadcast the ‘elected’ event to all nodes.

```prolog
:- table(node(Id, Address), [key(Id), key(Address)]).

election :-
    this_node(MyAddress),
    sendall(Addr, (node(Id, Addr), Id > MyId),
        election(MyAddress)),
    expect(ok, timeout, 3000).

event(election(Addr)) :-
    send(Addr, ok),
    election.

event(elected(Id)) :-
    log('Elected: ~p', [Id]).

timeout :-
    this_node(MyAddress),
    node(MyId, MyAddress),
    log('I am the leader: ~p', [MyId]),
    sendall(Addr, node(_, Addr),
        elected(MyId)).
```
4 DAHL data structures and storage

The central data structure of DAHL is that of terms. Terms are used to represent a wide variety of data types ranging from primitive ones (integers, floats, and strings) to arbitrary complex compound structures, e.g., tuples, lists, trees, queues, sets, and bags.

Data manipulation in DAHL is performed in a declarative fashion, e.g., data structures are traversed by first decomposing them into components and then iteratively descending into the appropriate component. Data structures are immutable, i.e., they cannot be modified in-place, and a program variable can be bound to a data value only once. The flow of data in DAHL programs can be organized in three ways: assigning a value to a variable, as parameter passing between procedures, and by storing/retrieving data tuples from the DAHL database.

The database is a storage facility that, since it persists during the program execution, can be used by developers for maintaining the application state. DAHL extends the storage facilities provided by standard Prolog implementations with automatic indexing for fast retrieval, akin to relational database management systems, as well as user defined key constraints that enforce uniqueness of data lookup by keys. This allows updates to the database to be done in a predictable, fast, and declarative way.

We included this feature in DAHL as it proved to be useful for the development of distributed systems in the P2 language, which was later confirmed by us while developing applications in DAHL.

The database organizes the stored data into tables. In order to use DAHL’s extensions, each table has to be declared. Note that standard Prolog tables can still be used without declaration, although without the benefits provided by DAHL (Prolog only provides limited indexing capabilities). The syntax for declaring tables is the following:

\[ :- \text{table}(\text{Template}, [\text{Options}]). \]

where Template is a term describing the name and number of fields in the table, and Options is a list containing terms of the form:

unique, key(Vars), or index(Vars).

The key options, as well as its special case unique, impose constraints on the combinations of tuples that can be stored on the table. Specifically for each assignment to the variables in the key, at most one tuple can be stored in the database. For example,

\[ :- \text{table}(f(X, Y), [\text{key}(X)]). \]
\[ :- \text{table}(g(X, Y), [\text{key}(X), \text{key}(Y)]). \]
\[ :- \text{table}(h(X, Y), [\text{unique}]). \]

declares \(f\) to behave as a function so that, for each value of the key \(X\), there is at most one corresponding value \(Y\). Similarly, \(g\) is defined to satisfy the constraints of a one-to-one mapping. Finally, the table \(h\) can store at most one tuple. Note that this is the special case of a key with no variables. The index options specify fields to be automatically indexed for fast retrieval. Fields declared with key are also indexed, but index does not impose any constraints on the number of tuples that can be stored for each assignment to the index variables.

5 DAHL clauses and operational semantics

DAHL borrows its syntax from Prolog and other similar languages in the context of constraint logic programming (CLP). Code is organized into clauses of the form:

\[ \text{Head} :\text{- Body}. \]

Such clause defines that the predicate in the Head should be evaluated according to the query-like expression defined by its Body. The Body is an arbitrary expression built from operators such as conjunction, negation, and conditionals, as well as calls to built-in or user defined predicates. The actual semantics used to evaluate the bodies of clauses is inherited from Prolog’s procedural semantics which allow the execution of operations with complex control flows, including iteration, recursion, backtracking and side-effects.

Broadly speaking, the semantics of evaluation can be described as follows. To evaluate a predicate, also known as a goal, the system searches for a clause declaration whose head unifies with the goal. If a match is found, the matching clause is activated and each of the goals on its body are themselves evaluated from left to right. If, at any time, the system fails to find a match for a goal, it back-
tracks undoing previous unifications, and reconsid-
ers alternative matches for the activated clause that
got rejected. For a more precise and detailed de-
scription, we refer the reader to the rich body of
literature on Prolog’s books and manuals [22,24].

We devote the rest of this section to describe the
additional features and extensions – and their se-
manics – particular to DAHL.

The control structure of a DAHL program, suit-
able for the development of distributed and reactive
systems, is driven by an event manager. The
event manager triggers the evaluation of clauses for
events received from the network, or caused by the
expiration of alarms. This is achieved by declaring
special clauses of the form:

\[\text{event}(\text{Head}) \leftarrow \text{Body}.\]
\[\text{alarm}(\text{Head}) \leftarrow \text{Body}.\]

These are respectively called event and alarm han-
dlers. An event handler is called automatically
when a corresponding event arrives from the net-
work. Alarm handlers are called when an associ-
ated timer expires. Alarms (that can be used, for
example, for periodic message retransmission) are
further described in the following section.

By selectively marking which clauses can be trig-
gered remotely, we ensure that no random local
clause can be triggered by a possibly malicious re-

ter remote host.

6 DAHL Runtime Implement-
tation

This section gives an overview of the current DAHL
runtime. It starts with a presentation of its imple-
mentation, which is then followed by the results of
a synthetic benchmark to assess the performance of
our implementation.

6.1 Implementation

DAHL is implemented using the SICStus Prolog
compiler [24]. It consists of a runtime library writ-
ten in Prolog (with around 460 lines), and an op-
imized networking back-end written in C (around
450 lines of code). The networking back-end uses
libevent [15] to efficiently dispatch the incoming
events from the network. The back-end also inter-
faces with the OpenSSL library in order to imple-
ment the cryptographic primitives in the language.

In order to interface with Prolog, the back-end uses
stubs that are generated automatically by the SIC-
Stus Prolog compiler.

6.1.1 Runtime

DAHL programs are interpreted directly by the
SICStus Prolog compiler, but under the DAHL run-
time control. The main program in execution is
part of the runtime, and a DAHL program’s code
is only called when an appropriate event arrives
from the network, or when a timer is triggered.

The runtime works by running a loop to receive
an event and dispatch it to the right handler. The
loop is implemented in Prolog, and the process to
wait for events is done by libevent. This library
performs a non-active wait for network and timer
events (if the underlying operating system supports
that).

Figure 3 shows the execution flow for process-
ing a message that arrives from the network (steps
1–4), and for a message that is sent from an appli-
cation (steps 5–7) in more detail. When a message
arrives from the network, the operating system dis-
patches it to libevent (step 1), which queues the
message. Then, when the DAHL runtime asks for
the next message, libevent picks one arbitrarily and
delivers it to the DAHL network back-end (step 2).
The DAHL network back-end then deserializes the
message and calls the runtime dispatcher (in Pro-
log) through a stub (step 3). Finally, the dispatcher
triggers an event to the corresponding event han-
dlter of the application (step 4). When a DAHL
application sends a message, the message is first
handed over to the DAHL runtime through a stub
(step 5). The runtime then serializes the message
and delegates the network transmission to the op-
erating system (step 6).

6.1.2 Optimizations

We implemented several optimizations in the
DAHL runtime to improve its performance. Here,
we present these optimizations in detail. The
deserialization of network messages was a CPU-

intensive operation since the SICStus Prolog com-
piler implements this operation in Prolog through
a complex process chain. Since each message sent
was serialized to a single atom, it led to an explo-
sion in memory usage because the SICStus Prolog compiler aggressively caches all atoms. We therefore implemented our own custom deserialization in C to improve performance. This resulted in a performance improvement of the deserialization function of about 70%.

As described before, the main loop is implemented in Prolog, and it calls a function in C that “produces” events through libevent, which are then dispatched from within the Prolog environment. After an event is dispatched and processed, the loop backtracks until the beginning of it. This provides an important advantage, which is that every event/alarm handler is executed in a “clean” environment, as all the garbage possibly left by a previous handler is discarded. Moreover, it improves the performance of the garbage collector (GC), as the SICStus Prolog compiler will delete most of such garbage when backtracking as an optimization, reducing the overhead of the GC. Our tests show that without this environment cleanup, the overhead of the GC would be noticeable (from 8% to 45%).

6.1.3 Network support

Currently all the network events are sent using the TCP protocol, which requires establishing a connection before the first contact. The DAHL runtime automatically establishes these connections when needed, and caches them indefinitely for future contacts. Other caching policies can be easily implemented in the runtime if required by the application. Moreover, although the TCP protocol establishes a bidirectional channel, TCP connections opened by the DAHL runtime are used unidirectionally. This is because it would require extra communication in order to reliably detect that a given input channel corresponds to the address of an outgoing message (including the port number), which would not be desirable for some applications (such as sensor networks, or any bandwidth restricted application).

6.2 Basic Performance Evaluation

To evaluate the performance of the DAHL runtime, we performed a test to compare the performance of P2, Mace, C, and DAHL. We performed a simple network ping-pong experiment. One of the machines (called a client) sends a small 20 byte ‘ping’ message to the other machine (called a server) which immediately responds with a small 20 byte ‘pong’ message. We use as many client machines as needed to saturate the server in order to measure its raw throughput. The measurement of the number of requests served per second is done at the server. The machines were connected by a gigabit switch with a round trip latency of 0.09 ms, and both the network and the machines were unloaded.

The results are presented in Table 2.

First, we note that the DAHL runtime outperforms P2’s performance. We believe that the reason behind P2’s poor performance is that the runtime of P2 is not yet optimized while DAHL uses SICStus Prolog 4 [24] compiler that has been extensively optimized. Second, DAHL is as fast as Mace. However, given that Mace is a restricted form of C++, it can exploit powerful C++ compiler optimizations. For example, with the ‘-O2’ set of optimizations of gcc 4.1, Mace’s performance improves by 60%. As an upper bound on the performance, we also present the performance of a C implementation (using the high-performance networking library libevent [15]) and note that all systems that strive to improve the analysis capability are an order of magnitude slower than it.

Finally, we also note that the SICStus Prolog compiler does not perform any significant code optimizations. As an experiment, we modified the part of the code of the server that updates the number of served requests so that it did not use dynamic tuples nor DAHL’s add built-in predicate. The first version used a single Prolog blackboard fact (i.e. updated using bb_get and bb_put predicates), and the performance increased by 27%. The second version used a function in C with a native integer to store and increment the value. This solution increased the performance by 37% over the original
Table 2: Basic performance comparison between P2, DAHL, Mace and C (in requests per second).

<table>
<thead>
<tr>
<th></th>
<th>P2</th>
<th>DAHL</th>
<th>DAHL w/-Oh</th>
<th>Mace</th>
<th>Mace w/-O2</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>230</td>
<td>14,000</td>
<td>19,220</td>
<td>14,221</td>
<td>21,937</td>
<td>142,800</td>
</tr>
</tbody>
</table>

solution to 19,220 requests per second (denoted as -Oh – human optimized – in the table above). This experiment shows that the performance of DAHL could be improved with some simple code optimizations, which were not implemented due to time constraints.

7 Chord

In this section, an implementation of Chord [23] in DAHL is evaluated. Chord is a popular peer-to-peer (P2P) distributed hashed table (DHT), also called a P2P overlay.

The purpose of this evaluation is twofold. First, we wanted to measure a networking driven application performance by using application specific benchmarks. For this purpose, we ran the standard benchmarks for P2P overlays. Second, we wanted to compare the performance and succinctness of our implementation with the reference implementation (in C) and with other implementations in state-of-the-art languages for distributed systems. Our implementation of Chord implements all the features contained in the original paper [23], but using recursive queries and 160-bit identifiers.

To the best of my knowledge, P2 is the only state-of-the-art language for the implementation of distributed systems to offer an implementation of Chord. Therefore we compare our results only with P2. To compare with the P2 Chord implementation, we obtained the latest release of P2 at time of writing. Unfortunately, we were unable to get P2’s Chord implementation running in our local setup. We therefore cite results from their paper [14].

7.1 Evaluation

7.1.1 Setup

We used Modelnet [25] to emulate a GT-ITM transit-stub topology [4] consisting of 100 to 500 stubs and ten transit nodes. The stub-transit links had a latency of 2 ms and 10 Mbps of bandwidth while transit-transit links had a latency of 50 ms and 100 Mbps of bandwidth. We used 10 physical hosts, each with dual core AMD Opteron 2.6 Ghz processor with 8 GB of RAM, running Linux kernel version 2.6.24. We ran 10 to 50 virtual nodes on each physical node, producing a population of 100 to 500 nodes. In each experiment neither the CPU nor the RAM were the bottleneck. This setup reproduces the topology used by the P2 experiments in [14], although they used Emulab [28].

We have also ran the same experiments in Emulab, but with a smaller set of nodes (only ten), because it was not possible to reserve more machines in a timely fashion. Therefore we do not present the results from this experiment, although they were similar with the the ones obtained with Modelnet.

7.1.2 Static Membership

Our first goal was to see if the DAHL implementation of Chord met the high-level properties of the protocol. We have first evaluated our implementation by performing 10,000 DHT lookup requests generated from a randomly chosen node in the network for a random set of keys. The lookups were generated after waiting for five minutes after the last node joined in order to let the network stabilize. The nodes joined the overlay sequentially using a random landmark.

In Figure 4, we present the cumulative distribution of latency incurred to receive the response to the lookup requests with 100 and 500 nodes. The results are comparable or better than the published results for P2’s Chord [14].

In Figure 5 we present the frequency distribution of the number of hops taken to complete the lookups. As expected, the average number of hops taken by each query is around \( \log N \) and the maximum number of hops is under the theoretical limit of \( \lceil \log N \rceil \).
7.1.3 Dynamic Membership

Our implementation of Chord in DAHL also handles churn. In this experiment we used 500 nodes, each one maintaining four successors and performing finger fixing every 10 seconds and successor stabilization every 5 seconds. This configuration is similar to the setup of P2’s Chord. We generated artificial churn in our experiment by killing and joining nodes at random with different session times by following the methodology presented in [19]. Figure 6 presents our results for three churn rates. We found that with 30 minutes of session time, our implementation delivers approximately 66% of lookup consistency while with 60 minutes of session time, we obtain lookup consistency of 96%. While these results are lower than the consistency results for P2’s Chord (they achieve 84% of lookup consistency for 32 minutes and 97% of consistency for 64 minutes of session time), our implementation can be further tuned to improve the lookup consistency by, for example, implementing retransmission of queries on timeout.

As our implementation of Chord does not perform retransmission of timed-out queries, it is normal that our consistency will be lower than MIT’s original implementation. Moreover, as their implementation is iterative and ours is recursive, the results should not be compared directly.

7.1.4 Code Size and Effort

Our implementation of Chord is comparable in size to the P2 implementation in terms of lines of code (LoC). DAHL’s Chord is implemented in 215 LoC while P2’s is implemented in 211 LoC. These sizes are an order of magnitude more succinct than MIT’s reference implementation in C.

Chord was implemented in DAHL in one week by one person, who had implemented the protocol before in C#. During that period, we also did some simple tests in our cluster (like testing the result of lookups, and graphically displaying the membership graph). The deployment of Modelnet and the churn experiments took another week.

7.2 Summary

Our results show that our implementation of Chord in DAHL covers the major algorithmic aspects of the Chord protocol and that its performance is competitive with P2’s Chord. In particular, our implementation shows an average hop count of $\log_2 N$ for a lookup, and maintains a lookup consistency of 96% for 60 minutes of session time, which is similar to other implementations. Moreover, in terms of code size, our implementation is an order of magnitude smaller than the reference implementation, and about the same size as P2’s implementation.

In summary, DAHL delivers good performance in the benchmarks of a network protocol. The implementation of this protocol is also simple and succinct.
Model checking is a prominent way to verify the correctness of software and hardware designs (e.g., [1, 7–9]). However, model checking techniques suffer from the fundamental problem of state space explosion during exploration. Scalability of model checking for large, real systems thus remains a challenge.

In this section we evaluate a distributed software model checker, D’ARMC, based on the sequential ARMc model checker [17]. D’ARMC is essentially CPU-bound.

8.1 Evaluation

In this section the results of running D’ARMC on a set of benchmarks are presented. The benchmark suite is composed by difficult examples from the transportation domain and a standard hybrid system example, gasburner. Our benchmarks are automata-theoretic models compiled from complex specifications that describe communication, timing, and data manipulation aspects.

The tests were run on a cluster of AMD Opteron 252 (2.6 Ghz) machines, with 3 GB of RAM, 64 KB of L1 caches, and 1 MB of L2 cache each, running Linux kernel version 2.6.24. The computers were connected through a gigabit LAN with an average RTT of 0.14 ms.

Figure 7 presents graphically the speedups obtained in each test, as well as the median.

![Figure 7: Speedup of D’ARMC.](image)

It is worth noting that even with efficiency rates that are not great (the median is around 50% when refining just one counterexample), the running time is still dramatically decreased. For example, running the larger_scale1_ub test sequentially takes more than 2 days, while D’ARMC takes only a little more than 2 hours. This reduction is certainly a big win.

9 Conclusions

In this paper I’ve presented DAHL, a new programming system to simplify the construction of distributed programs. By providing a high-level language and an efficient runtime environment, DAHL enables the development of more succinct, albeit clear, programs, while enabling efficient execution and static analysis.

The benefits of DAHL are exemplified by building two representative distributed applications, such as Chord, a network-driven protocol, and D’ARMC, a CPU-bound application. To the better of my knowledge, D’ARMC is the world’s first scalable distributed software model checker, and is implemented in DAHL.

The applications implemented in DAHL were tested in a local cluster, with and without Modelnet, as well as in Emulab, which shows the high reliability and flexibility of DAHL.

References


