



Melange

Creating a “Functional” Internet

CS 6V81

Michael Hoglan

4/2/2008



Present Situation

- Presently most internet protocols are implemented in type-unsafe languages (C / C++)
- Protocols range from
 - simple (Ethernet, IPv4, ICMP and TCP)
 - complex (SSH, DNS and BGP)
- Vulnerable to security and reliability problems, most of which would be avoided in type-safe implementation
- type-safe solutions have been avoided due to their performance degradation (FoxNet)



Proposed Solution

- Combine techniques to overcome performance degradation and maintain type safety. Best of both worlds approach.
 - strong static typing
 - generative meta-programming
- Objective Caml (OCaml) provides strong static typing
- MPL provides generative meta-programming
- Technique introduces no performance penalty



Objective Caml (OCaml)

- Comes from the ML family of languages
- strong static typing
- Mix of programming styles; functional, imperative and object oriented
- Uses runtime bounds checking to guarantee type and memory safety
- Supports unsafe functions and external C bindings, but does not guarantee safety when used
- Uses single threaded fast garbage collector



Objective Caml (OCaml)

- Currently is used in several projects
 - CIL
 - Ensemble
 - Microsoft's Terminator
- Attractive due to its lightweight run time and its simplicity
- Provides a greater level of stability



Objective Caml (OCaml)

- GC splits heap into minor and major heap
- Minor heap contains small and short-lived objects
- Major heap contains large and longer-lived objects
- mark-and-sweep cleans out the minor heap when full and remaining objects are moved to major heap
- Ideal for network design
- Can be triggered through API



Meta Packet Language (MPL)

- Domain-specific language to describe binary network protocols
- Supports bi-directional to create / receive packets
- Acts as a meta-compiler and outputs OCaml
- Provides zero-copy capabilities
- Not human readable, but provides external interface
- Supports custom field types to allow more complex expressions (e.g. DNS)



Meta Packet Language (MPL)

- Described by Extended BNF grammar in paper
- Utilizes three possible types
 - wire types
 - MPL types
 - language types (OCaml)
- wire types map to MPL types map to language types
- classify keyword allows switch like behavior for basing decisions
- variant keyword allows enumeration behavior for the external interface



Meta Packet Language (MPL)

```
main → (packet-decl)+ eof
packet-decl → packet identifier [ (packet-args) ] packet-body
packet-args → { int | bool } identifier [ , packet-args ]
packet-body → { (statement)+ }
statement → identifier : identifier [var-size] (var-attr)* ;
           | classify ( identifier ) { (classify-match)+ } ;
           | identifier : array ( expr ) { (statement)+ } ;
           | ( ) ;
classify-match → '!' expr : expr [when ( expr )] -> (statement)+
var-attr → variant { ('!' expr {→ | ⇒} cap-identifier)+ }
           | { min | max | align | value | const | default } ( expr )
var-size → [ expr ]
expr → integer | string | identifier | ( expr )
       | expr { + | - | * | / | and | or } expr
       | { - | + | not } expr
       | true | false
       | expr { > | >= | < | <= | = | .. } expr
       | { sizeof | array_length | offset } ( expr-arg )
       | remaining ( )
```

Figure 2: EBNF grammar for MPL specifications



Meta Packet Language (MPL)

```
packet ipv4 {
  version: bit[4] const (4);
  ihl: bit[4] min (5) value (offset (options) / 4);
  tos_precedence: bit[3] variant {
    0 → Routine | 1 → Priority
    2 → Immediate | 3 → Flash
    4 → Flash_override | 5 → ECP
    6 → Inet_control | 7 → Net_control
  };
  delay: bit[1] default (false);
  throughput: bit[1] default (false);
  reliability: bit[1] default (false);
  reserved: bit[2] const (0);
  length: uint16 value (offset (data));
  id: uint16;
  reserved: bit[1] const (0);
  dont_fragment: bit[1] default (0);
  can_fragment: bit[1] default (0);
  frag_off: bit[13] default (0);
  ttl: byte;
  protocol: byte variant {
    1 → ICMP | 2 → IGMP | 6 → TCP | 17 → UDP};
  checksum: uint16 default (0);
  src: uint32;
  dest: uint32;
  options: byte[(ihl × 4) - offset (dest)] align (32);
  header_end: label;
  data: byte[length - (ihl × 4)];
}
```



OCaml Interface

- Generated OCaml code by the MPL compiler
- Each packet is represented as a string
- packet environment record allows a view into the data which can be cheaply copied
- payload data is shared across all packet environments and is not copied
- Eliminates need to write error prone, tedious, low level code
- Allows additional low level code to be used if necessary
- packet suspension allows writing of packet in one operation



Performance (ICMP)

- Simple comparison
- Compare two Melange implementations of ICMP with lightweight IP (lwIP)
 - copying version
 - MPL version (zero copy)
- lwIP is mainly used in embedded systems
- ping uses ICMP echo request / response packets
- MPL version is almost identical to lwIP, with a slight performance hit due to calculating checksum
- Using a 'reflecting' version of the OCaml implementation that modifies the checksum in place, matches lwIP performance.

Performance (ICMP)

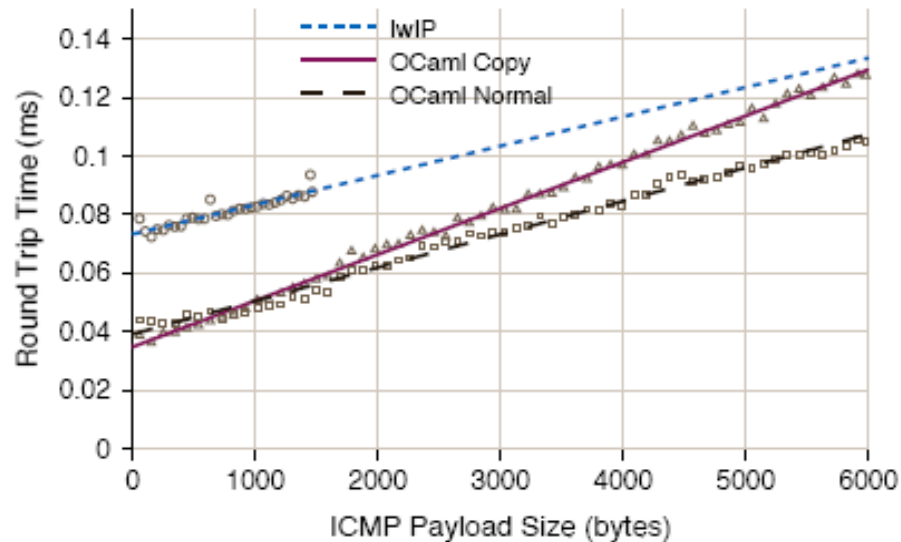


Figure 4: Latencies for lwIP vs OCaml “functional” version (*OCaml Copy*) which copies data and a normal MPL version (*OCaml Normal*) (*lower gradient is better*).

Performance (ICMP)

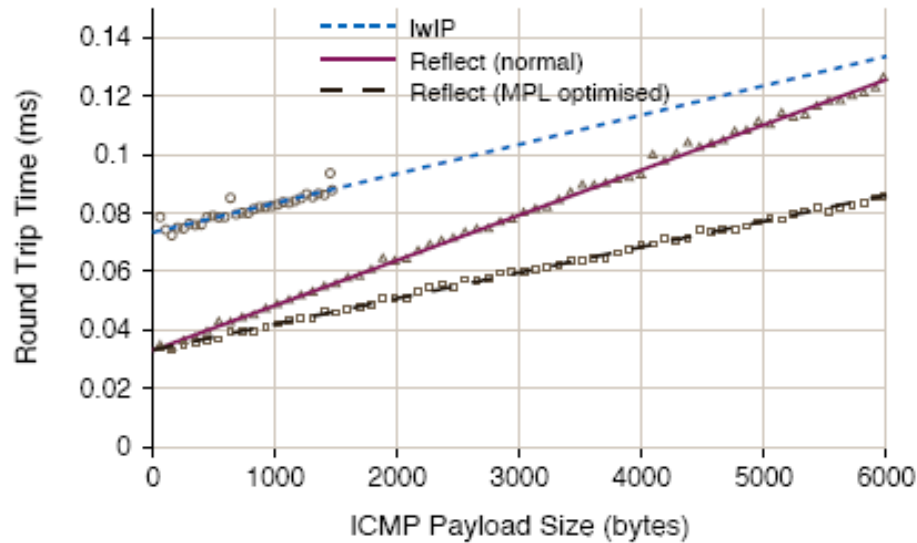


Figure 5: Latencies for lwIP vs the OCaml “reflector” with MPL bounds optimisation off (*Reflect normal*) and on (*Reflect MPL optimised*). The MPL optimised version is type-safe OCaml and as fast as lwIP.



MLSSH

- SSH implemented in the Melange framework
- Composed of several layers
 - transport layer
 - authentication layer
 - data channels
- Protocol has more complexities than ICMP
 - key exchange
 - negotiation / re-keying
 - authentication modes
 - dynamic channel multiplexing
- Only external bindings used were the ones for encryption algorithms (e.g. SHA-1, AES-192 provided by OCaml Cryptokit library)

MLSSH

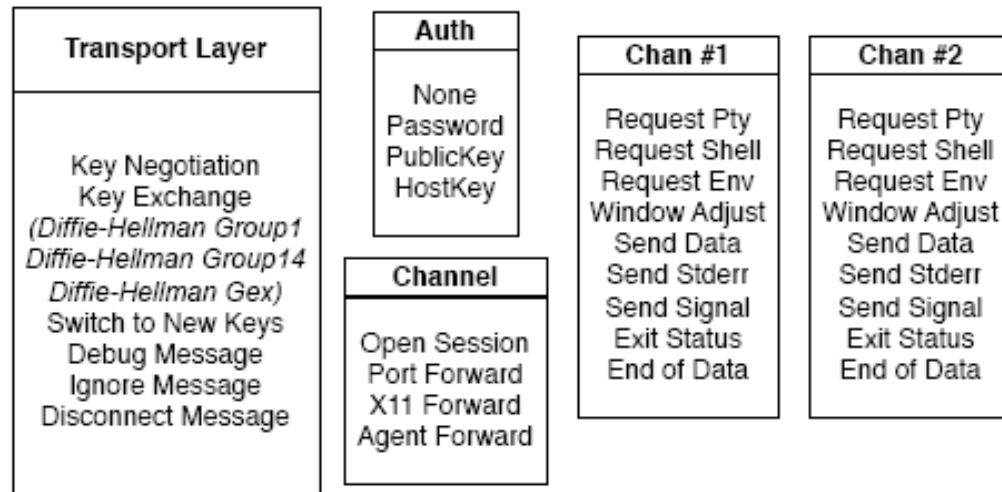


Figure 6: Various layers of the Secure Shell v2 protocol: a global transport, authentication and channel layer, and local channel states.

MLSSH

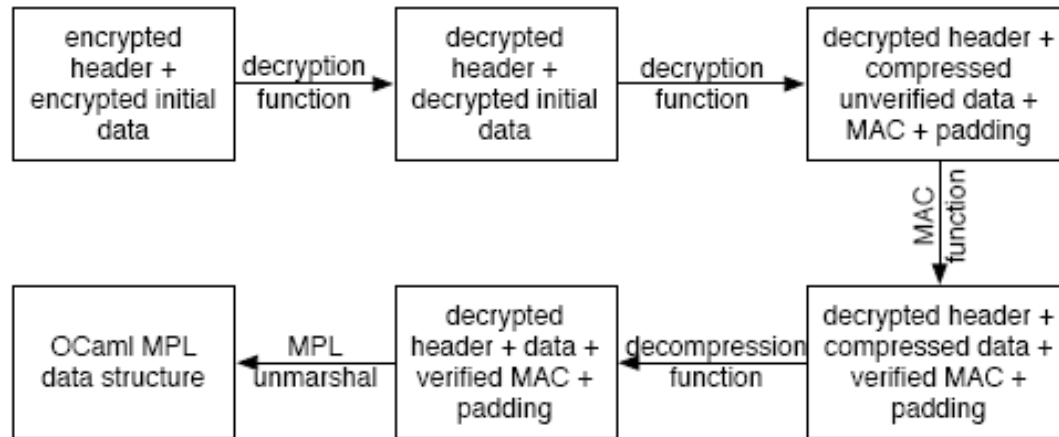


Figure 7: Illustrating the complex data flow of SSH wire traffic to plain text payload that can be parsed using MPL.



MLSSH

- Packets are composed from different stages
- Early implementations did not use MPL and thus did not support zero copy
- Protocol quirks were handled with hand written OCaml interfacing with MPL generated code. (e.g. encryption algorithms)



Performance (MLSSH)

- Various size files transferred 100 times across SSH connection; at least 10GB in all
- MLSSH slightly faster than OpenSSH
- OpenSSH exhibits some jitters
- MLSSH hampered in encryption tests due to OCaml AES implementation
- Garbage collection did not burden MLSSH
- Analysis of the internals of OpenBSD malloc / free show that manual memory management is just as complex as garbage collector routines

Performance (MLSSH)

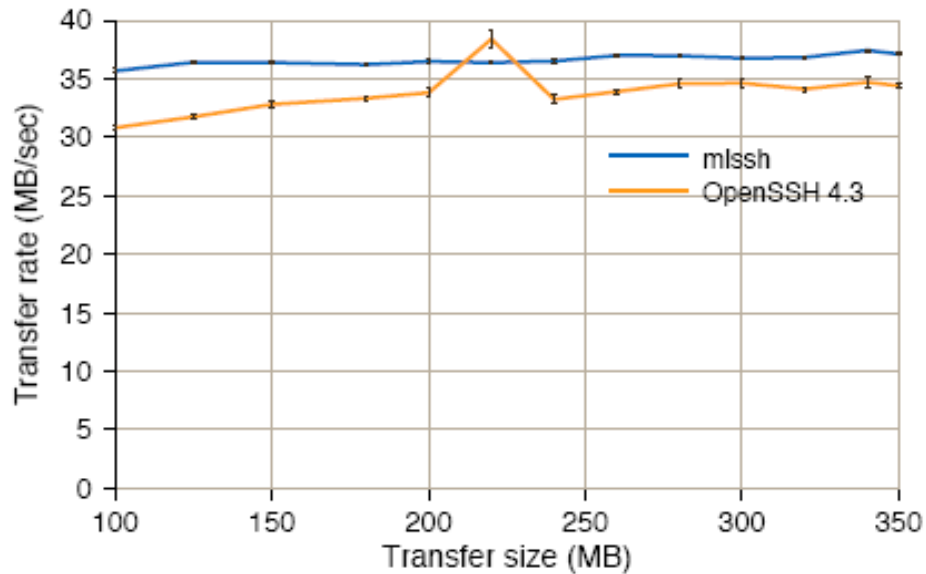


Figure 8: Throughput of OpenSSH vs MLSSH with encryption and message hashing disabled (*higher is better*).

Performance (MLSSH)

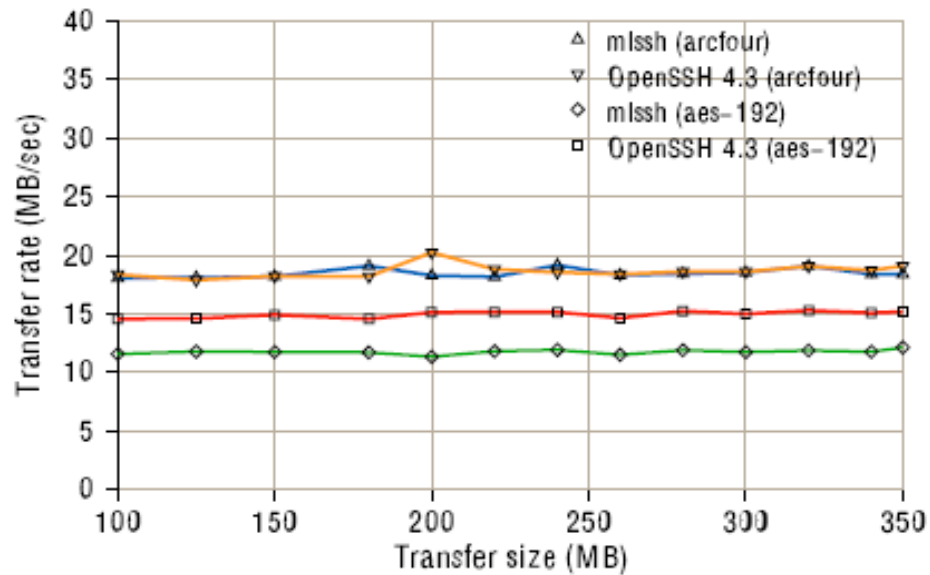


Figure 9: Throughput of OpenSSH vs MLSSH using stream and block ciphers (*higher is better*).

Performance (MLSSH)

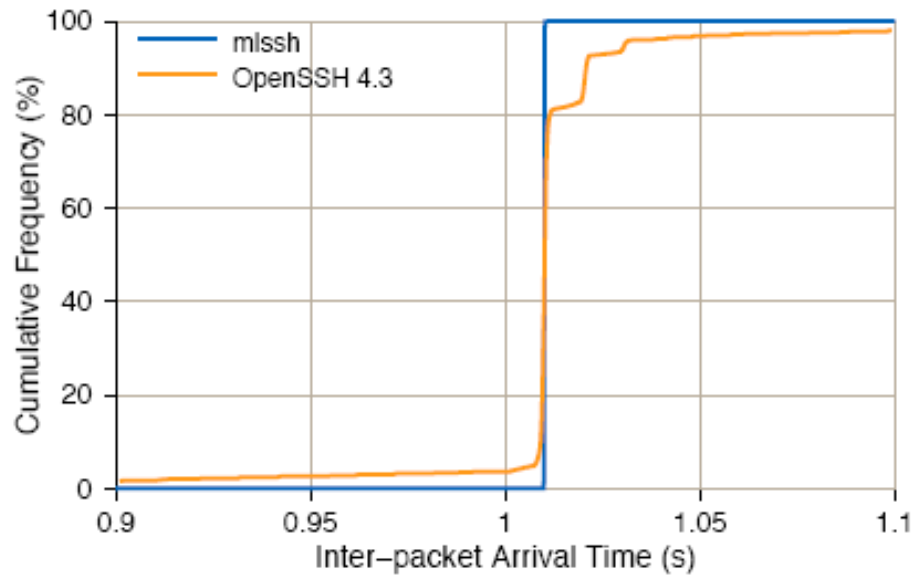


Figure 10: Cumulative Distribution Function of inter-packet arrival times of OpenSSH and MLSSH.



DNS

- Protocol designed to be low latency low overhead for resolving domain names
- BIND serves over 70% of DNS domains
- 99% of DNS servers are written in C
- Historically has been susceptible to security vulnerabilities
- Difficult to implement securely and safely due to compression scheme



DEENS

- DNS server written in OCaml / MPL
- Custom types were implemented in OCaml to handle the compression scheme
 - dns_label
 - dns_label_comp
- DNS packets can be processed in single pass
- Further optimized with memoisation, caching the result response packet in a hash instead of re-creating it constantly



Performance (DEENS)

- Non-memoisation
 - DEENS is 10% faster in queries per second
 - Slightly faster than BIND in cumulative latency
- Memoisation
 - DEENS is 2x faster in queries per second

Performance (DEENS)

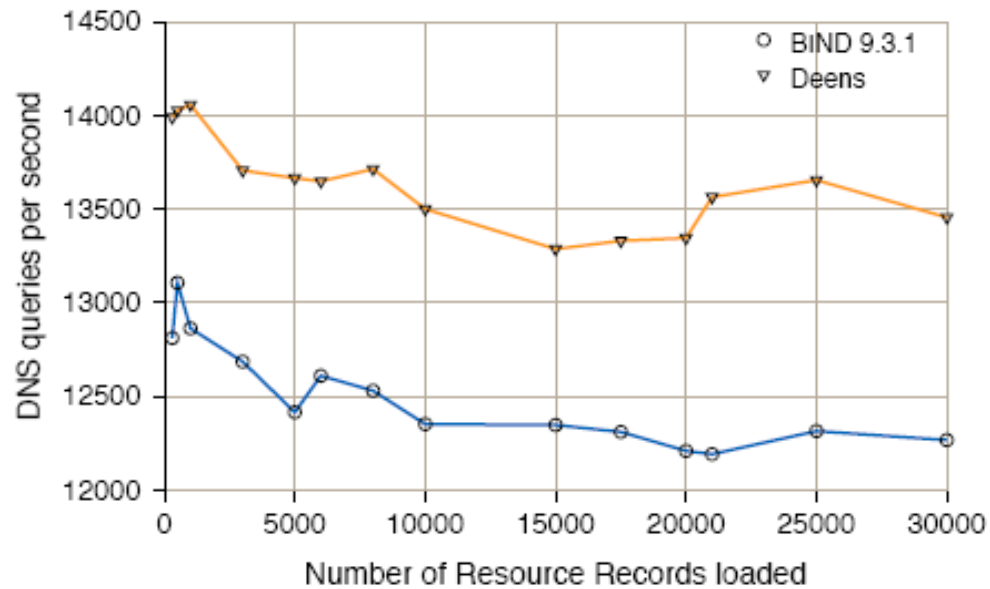


Figure 12: Throughput of BIND vs DEENS with random Zipf-distribution query sets (*higher is better*).

Performance (DEENS)

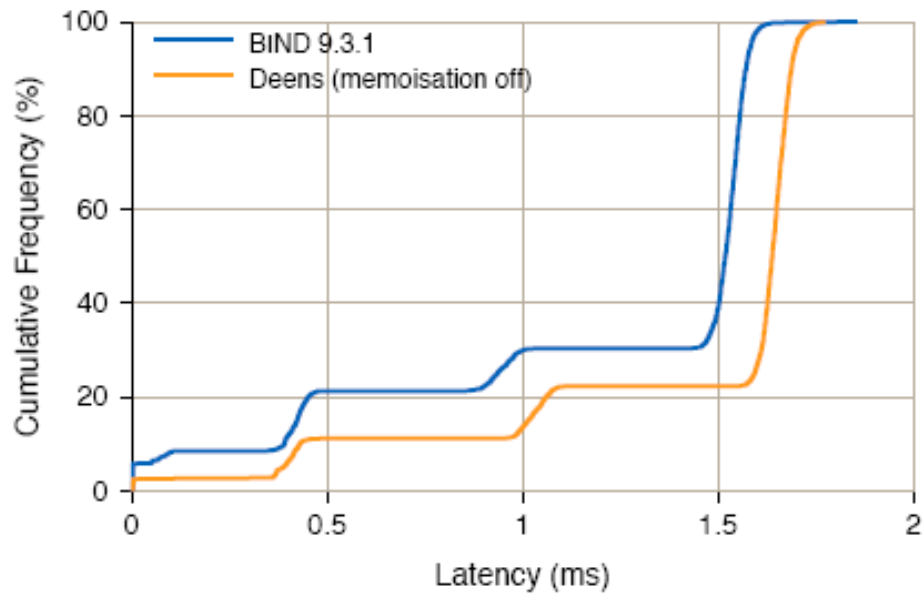


Figure 13: Cumulative Distribution Function of BIND vs DEENS latencies with loaded servers (*lower is better*).

Performance (DEENS)

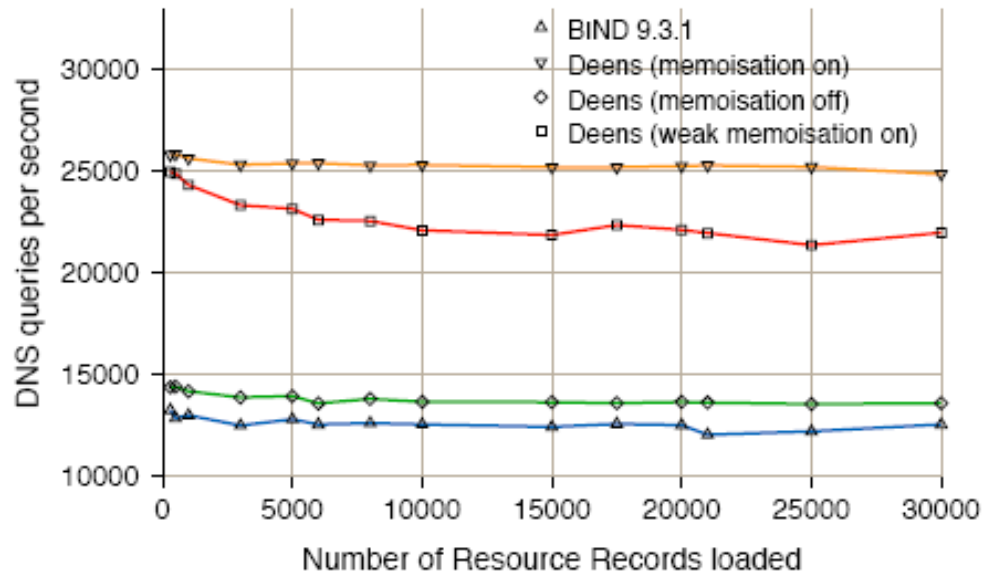


Figure 14: BIND vs DEENS throughput with the strong and weak memoisation optimisations with random Zipf-distribution query sets (*higher is better*).



Profiling

- DEENS and BIND should both exhibit different run time profiles due to their implementation methodologies
- Categorize time
 - System calls
 - Network packet handling
 - Libraries
 - Memory management
 - OCaml run-time library
 - Data structure management
 - Other code



Profiling

- BIND / DEENS spend most of its time in data management / network packet creation.
- BIND spends more time creating packets since DEENS uses the more efficient MPL generated code
- DEENS spends more time in memory management due to garbage collection.
- Memoised versions of DEENS spend even more time in memory management due to the constant garbage collection
- Both spend the same amount of time in system calls and external libraries.
- Memoised DEENS saturates a gigabit line, performing over 3x more query responses per second than DEENS / BIND

Profiling

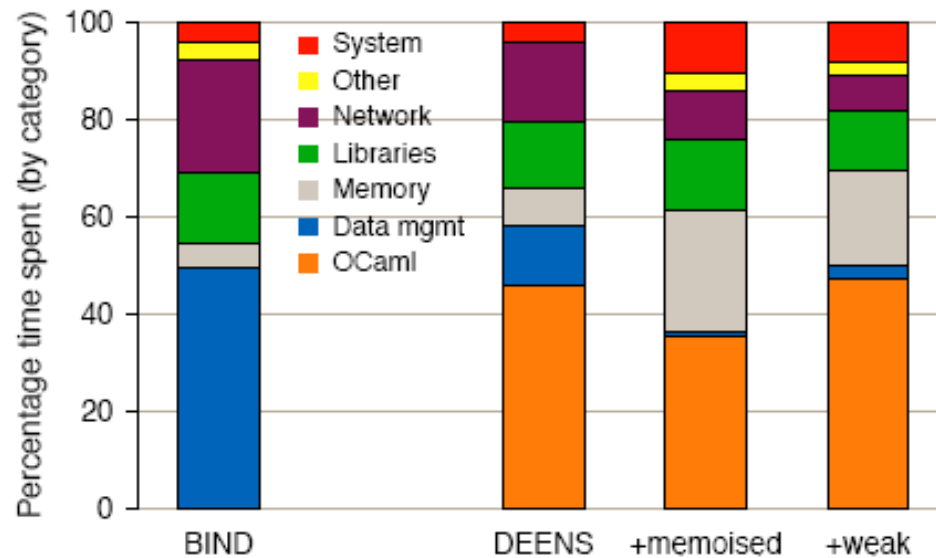


Figure 15: Normalised profiling results for the DNS servers, showing how each application spends its time serving queries.



Profiling

- Compare the LOC of MLSSH / OpenSSH and BIND / DEENS
- All were run through their pre-processors and for BIND / OpenSSH all platform portability code was removed from counts.
- OpenSSH is 3x larger than MLSSH
- BIND is 50x larger than DEENS
- DEENS is missing some features of BIND but that still does not account for the size difference. (e.g. DNSSEC)



Configuration

- Since Melange itself is composable and separated, the configuration itself becomes so to
- Configuration can be represented by functional objects that are implemented by the different applications
- This encourages re-usability and unifying configuration among different modules such as logging and network setup



Related Work

- FoxNet implemented TCP/IP in ML; however it had serious performance issues
- Ensemble uses OCaml to create simple ‘micro-protocols’;
- Ensemble introduces direct marshalling which exhibits a higher level of trust than MPL does. This possibly allows corruption of the heap due to malicious nodes.
- Other projects such as PacketTypes and Prolac implement low level protocols but they still output C instead of a high level language such as OCaml



Conclusions

- MPL mixed with OCaml provides the Melange framework has shown it is a very viable solution to writing low level internet protocols in a high level type safe language.
- Implementations were not only more efficient, they were more concise in size which should provide more reliability.