System Knowledge Base (SKB)

Adrian Schüpbach
Systems Group, ETH Zürich

Problem

- Increasingly complex hardware
  - Non-uniformity
  - Core and hardware diversity
  - System diversity
  - Complex hardware-given constraints
  - → cannot hardcode or hand-tune algorithms and systems
- Managing heterogeneity
  - Gathering information about current system
  - Intelligent allocation of resources
  - Meet hardware-given constraints
- Approach: System knowledge base (SKB)
  - Describe prefered solution in a declarative way
  - Think about high-level algorithms to manage heterogeneity
  - Let SKB compute a result

SKB

- very rich representation of hardware
  - central information store
  - queried by OS servers and applications
- introduce powerful reasoning engine
  - algorithms
  - optimizations
- expected benefits
  - more flexible and expressive policies
  - better adaptivity at runtime
  - code simpler and cleaner

SKB

Implementation

- OS service in separate domain in Barrelfish
- ECLiPSe-CLP
  - Prolog database
  - constraint logic programming (CLP) reasoning engine
- populated with lots of informations about the system
- Interface library for clients written in C

SKB

Examples

- simple examples:
  - NUMA:
    - facts about NUMA regions and core - NUMA affinity
    - query: find the address range closest to core X
  - cores:
    - facts about CPU packages, cores, hyperthreads and cache-sharing
    - query: find two cores (not threads) which share a cache
- On top of these simple queries more advanced ones

Outline

- Overview of how to create a SKB-based application
- How to imlement algorithm
- Overview of the available information

Create an application

Steps

1. Create program
2. include the skb.h file
   - `#include <skb/skb.h>`
3. link libskb to program
4. use C interface to
   - add information
   - query information
   - run algorithms within SKB

Create an application

Create a minimal program

Minimal application in test.c:

```
#include <skb/skb.h>

int main(int argc, char **argv)
{
    errval_t err = skb_client_connect();
    if (err_is_fail(err)) {
        DEBUG_ERR(err, "connection to SKB failed");
        ... some useful error handling ...
    }
    return 0;
}
```

Create an application

Create a minimal program

Link libskb to the application: Define source file and libskb in Hakefile

```
[ build application { target = "test",
                      cFiles = [ "test.c" ],
                      addLibraries = [ "skb" ]
                    }
]
```

Interacting with the SKB

- possibly add facts by calling into libskb
  - `int skb_add_fact(char *fmt, ...);`
- call algorithm via C interface
  - `int skb_execute_query(char *fmt, ...);`
- read result
  - `errval_t skb_read_output(char *fmt, ...);`

Interacting with the SKB

Add information to the SKB

```
#include <skb/skb.h>

int main(int argc, char **argv)
{
    errval_t err = skb_client_connect();
    ... connection error handling ...
    // gather information
    int nr_pci_roots = ...;
    err = skb_add_fact("simple_fact.");
    ... error handling ...
    err = skb_add_fact("nr_of_pci_root_complexes(%d).",
                       nr_pci_roots);
    return 0;
}
```

Interacting with the SKB

Load the ECLiPSe-Algorithm and call main goal

```
#include <skb/skb.h>

int main(int argc, char **argv)
{
    ... connect to to SKB and add facts ...
    char input[] = "input_text";

    err = skb_execute_query("[test_algo], "
                            "test_algo(%s, Output),"
                            "write(Output).", input);

    return 0;
}
```

Interacting with the SKB

Read the result: Single element

```
#include <skb/skb.h>

int main(int argc, char **argv)
{
    char textoutput[64]; int number;

    err = skb_execute_query(...);
    ... error handling ...
    err = skb_read_output("element(%[a-z], %d).",
                          textoutput, &number);
    ... error handling ...

    return 0;
}
```

Interacting with the SKB

Read the result: List

```
#include <skb/skb.h>

int main(int argc, char **argv)
{
    char textoutput[64]; int number;
    struct list_parser_status status;
    ... execute query ...
    skb_read_list_init(&status);

    while(skb_read_list(&status, "output(%d, %[a-z])",
                                 &number, textoutput) {
    }
    return 0;
}
```

Interacting with the SKB

```
#include <skb/skb.h>

int main(int argc, char **argv)
{
    errval_t err = skb_client_connect();
    if (err_is_fail(err)) {
        DEBUG_ERR(err, "connection to SKB failed");
    }
    // gather information
    int nr_pci_roots = ...;
    err = skb_add_fact("simple_fact.");
    ... error handling ...
    err = skb_add_fact("nr_of_pci_root_complexes(%d).", nr_pci_roots);
    ... error handling ...

    char input[] = "input_text";
    err = skb_execute_query("[test_algo], test_algo(%s, Output), write(Output).", input);

    char textoutput[64]; int number;
    err = skb_read_output("element(%[a-z], %d).", textoutput, &number);

    err = skb_execute_query("[test_algo], test_algo_list(%s, Output), write(Output).", input);
    struct list_parser_status status;
    skb_read_list_init(&status);

    while(skb_read_list(&status, "output(%d, %[a-z])", &number, textoutput) {
        ... do something with "number" and "textoutput" ...
    }
    return 0;
}
```

Interacting with the SKB

Status

- Know how to implement SKB-based application
- Need to know how to implement algorithm

Algorithms

Steps

1. Create separate file containing ECL^{i}PS^{e}-CLP code
   - Store the file under usr/skb/programs
   - Will be added to ramfs
2. Implement algorithm based on stored facts and input parameters

Algorithms

Implementation

- ECLiPSe-CLP is a programming language
- Create data structure
  - Lists
  - Trees
  - Variables
- Implement algorithm
  - As declarative as possible
  - Use data structure
  - Use input values
  - Use stored facts

Algorithm

Simple example: Get list of address regions of a type passed by application

File test_algo.pl:

```
test_algo_list(Input, Output) :-
    % create list with "region(Base, Size)" elements
    findall(region(Base, Size),
            (
               % access stored facts
               mem_region_type(Type, Input),
               memory_region(Base, _, Size, Type, _)
            ),
           Output).
```

Algorithm

Simple example: Get list of address regions of a type passed by application

Call from C application:

```
err = skb_execute_query("[test_algo], "
                        "test_algo_list("ram", Output), "
                        "write(Output).");
```

Output:

```
Output = [region(29880320, 4096),
          region(29884416, 524288),
          region(30408704, 1048576),
          region(31457280, 2097152),...]
```

Algorithm

Simple example: Get list of address regions of a type passed by application

Read output in C application:

```
uint64_t base, size;
struct list_parser_status status;
skb_read_list_init(&status);

while(skb_read_list(&status, "region(%lu, %lu)",
                    &base, &size) {
    ... do something with "base" and "size" ...
}
```

Current SKB "schema"

No fixed schema

- Not a fixed schema like in a database
- Prolog/ECL^{i}PS^{e} support facts with same name and different arities
- can always add arbitrary facts
- Already many facts in the SKB (if pci and datagatherer are running)

Current SKB "schema"

```
apic(ACPI_ProcessorID, APICID, Availability). % 1 = Yes, 0 = no
bridge(pcie|pci, addr(Bus, Dev, Fun), VendorID, DeviceID, Class, SubClass, ProgIf, secondary(Sec)).
device(pcie|pci, addr(Bus, Dev, Fun), VendorID, DeviceID, Class, SubClass, ProgIf, IntPin).
interrupt_override(Bus, SourceIRQ, GlobalIRQ, IntiFlags).
rootbridge_address_window(addr(Bus, Dev, Fun), mem(Min, Max)).
bar(addr(Bus, Dev, Fun), BARNr, Base, Size, mem|io, (non)prefetchable, Bits (64|32)).
fixed_memory(Base, Limit).
apic_nmi(ACPI_ProcessorID, IntiFlags, Lint).
memory_region(Base, SzBits, SzBytes, RegionType, Data).
currentbar(addr(Bus, Dev, Fun), BARNr, Base, Limit, Size).
pir(Source, Interrupt).
ioapic(APICID, Base, Global_IRQ_Base).
prt(addr(Bus, Dev, _), Pin, Source).
rootbridge(addr(Bus, Dev, Fun), childbus(MinBus, MaxBus), mem(Base, Limit)).
mem_region_type(Nr, Type).
memory_affinity(Base, Length, ProximityDomain).
cpu_affinity(APICID, LocalSAPIcEID, ProximityDomain).
tlb(APICID, level, data|instruction, AssociativityCode, NrEntries, PageSize).
cache(name, APICID, level, data|instruction, size, AssociativityCode, LineSize, LinesPerTag).
associativity_encoding(vendor, level, AssociativityCode, Associativity).
cpu_thread(APICID, Package_ID, Core_ID, Thread_ID).
maxstdcpuid(CoreID, MaxNrStdFunctions).
vendor(CoreID,Vendor (amd|intel).
message_rtt(StartCore, DestCore, Avg, Var, Min, Max).
nr_running_cores(Nr).
```

Further information

- source code
  - populating the SKB
    - `usr/skb/measurement/cpuid.c`
    - `usr/pci/pci.c`
  - querying the SKB
    - `lib/memusb/usb_mem.c`
    - `usr/fish/fish.c`
    - `usr/skb/testapps/map.c`
    - `usr/pci/pci.c`
  - available functions
    - `include/skb/skb.h`
- schema will be in doc directory
- tech-note as future work

Conclusion

- SKB facilitates dealing with complex hardware
- Easy to implement SKB-based application
- Implementing a correct and efficient algorithm might sometimes be a challenge