Towards versatile Models for Contemporary Hardware Platforms

Hendrik Borghorst, Karen Bieling and Olaf Spinczyk
Department of Computer Science 12
Technische Universität Dortmund, Germany
e-mail: {hendrik.borghorst, karen.bieling, olaf.spinczyk}@tu-dortmund.de

***Abstract*—The demand for computationally intensive workloads in the domain of real-time systems is growing which needs to be satisfied with more capable hardware. Cheap but powerful multi-core hardware seems to be a good solution but these processors often lack a good predictability. An operating system can hide measures to regain the predictability, like cache management but to do so a good knowledge of the hardware is required. A problem with these measures is that they are usually not portable and require a lot of work to adapt them to new platforms. It is desirable to generate the platform-specific code from abstract architecture descriptions to get a portable operating system that adapts itself to the specific hardware properties of modern hardware to provide a predictable execution environment.**

I. Introduction

Workloads within the real-time domain are getting more and more computationally intensive with the automotive industry pushing for autonomous cars, real-time face detection systems in security systems or power supply line monitoring for smart-grids. To meet this demand and at the same time reduce the cost of the hardware it is preferred to use cheap standard hardware with capable multi-core processors. But the low price for high performance computing power comes at the cost of loss of predictability.

These multi-core processors are usually designed to share resources to keep both the energy consumption and the price low. As a result of this, the timing behavior of these processors is not predictable and therefore they are not directly suitable for the use in real-time systems. The primary sources of unpredictability are caches [1], buses [2] and the main memory [3]. These sources of unpredictability have been in the focus of research for some time. Software-based control of the content of the shared caches has been proven to be an effective instrument to reduce the unpredictability of caches [4]. Rescheduling of memory accesses also was shown to be a viable mean to improve the memory access behavior [3].

These approaches can be used to reduce the unpredictability of modern hardware but often require special knowledge of the hardware and software a system uses. This could lead to an additional complexity for the system developers because they have to take into account on what hardware their code runs. One example is the alignment of data structures to cache line sizes. Instead these platform-specific measures should be handled by the operating system. In the past we presented an operating system concept that explicitly manages what data is in the cache, to get a more predictable system[5]. A problem with an approach like this is that it adds even more platform-specific code to the operating system which should be avoided. Instead we would like to write code that is normally platform-specific, in a new generic way to reuse it for all platforms. To do so we present an approach that uses an domain-specific language to describe an hardware architecture that can be used to generate code for low-level operating system functions like context switching, cache management, memory protection and other low-level functions that need to be written for every new hardware platform.

On the other hand an operating system also needs good information about the hardware it uses to fully utilize all the resources as good as possible. To achieve this the system needs a comprehensive model about the available resources and the timing behavior of a platform. An empirical approach to generate such a model can be used and the methods to do so can also be generated by the code generation. Profiling of a hardware architecture is used as a case study for this paper as it is usually a complex task because the profiling code has to be written in a low-level assembly language [6]. The model that is generated should provide essential information to the operating system at runtime and during the compilation to optimize it as much as possible.

In the following section we present an approach that utilizes a generic domain-specific language to describe hardware architectures with all their details needed to generate platform-specific code that can be used instead of manually written hardware-adaption code.

II. Approach

To specify an architecture we chose an approach with a domain-specific language (*DSL*). A language to model a hardware architecture, needs to be flexible enough to be able to specify current and upcoming architectures. This means that it should not have limitations, how the memory system of a architecture is structured. The language should be able to model a processor with multiple scratchpad memories for one processor core and a *NUMA*-based architecture just as well. To completely model a memory hierarchy it is also important to represent the interconnects between components like memories or processor cores correctly to ensure that the operating system can later take full advantage of measures to increase the performance and predictability of the hardware.

```
architecture ExampleArch {
 Memory RAM {}
 Memory L2Cache : RAM {}
 Memory Cache0 : L2Cache {}
 Memory Cache1 : L2Cache {}
 Processor CPU0 : Cache0 {}
 Processor CPU1 : Cache1 {}

 ISA {
  registers { R%[0..15] }
  instructions {
     add_const ADD: dest, arg, #arg const
     add_reg ADD: dest, arg, arg
    }
  }
}
```

Fig. 1: Example of an architecture description

Besides the memories, interconnects and computing units the architecture description needs to specify the instruction set architecture (*ISA*) of the available computing units. This ISA description is used for the code generator and contains details about the available registers, with the names used by the assembler for the architecture, and a basic set of assembly instructions. For heterogeneous architectures it is also possible to specify multiple ISAs for one architecture. So that different processor cores could use different ISAs.

An example representation of an architecture is shown by Figure 1. It consists of two processors which are each connected to a private cache, that is connected to a shared level-2 cache. The last item in the memory hierarchy is a main memory called `RAM`. The example architecture also included the register specification for the registers R0 to R15. The interconnects between multiple components are directly derived from the inheritances, for example in Figure 1 level-2 cache is connected to both the private `Cache0` and `Cache1`.

The `instructions` block includes all platform-specific assembly instructions needed for the abstract assembly language for the code generator. As an example Figure 1 only shows two instructions to add two values. Once with a constant and once with a value residing in another register. This language also allows to model a *NUMA*-based architecture by specifying multiple `RAM`-components that are only connected to one processor unit.

Figure 2 lists an example of a memory component. It describes a exemplary memory of the example architecture. To generate low-level operating system code it is necessary to specify some parameters that the operating system can use to optimize itself to the target architecture. These parameters include properties like the cache-line length (`minAccessSize`) or where a memory is mapped to in the address space.

In addition to the architecture description an abstract low-level development language needs to be defined. This language

```
RAM {
        wordLength: 4 // Bytes
        minAccessSize: 16 // Bytes
        startAddress: 0x40000000
        size: 2G
}
```

Fig. 2: Example of a memory component description

```
ram_benchmark {
  move(dest reg:0,arg %[bmStart_<BM>])
  move(dest reg:1,arg %[bmEnd_<BM>])

  jmp_mark(arg "loop_begin:")
  measure_start
  load(dest reg:3,src *reg:0)
  measure_end

  add_const(dest reg:1,arg reg:1,
  arg <WordLength>)
  cmp(arg reg:0,arg reg:1)
  cond_jump_lt(arg "loop_begin")
}
```

Fig. 3: Simple memory benchmark in abstract assembly code

is an abstract form of an assembly language that can be translated to platform-specific assembly code via the code generator. To do so the architecture description has to specify a minimal set of assembly instructions that are necessary for the code generator. The abstract assembly language can then be used to write low-level operating system code like context switching, cache flushing and time measurements in an abstract way so that it has to be done only once.

An example how to use the abstract assembly language is given with Figure 3. It depicts a memory read performance profiler. The profiling starts with the preparation of several constant values that are necessary to run the code like such as limits of the benchmark range. The next step is the creation of a label to create a loop over a certain benchmark range. Inside this loop is an abstract load instruction surrounded with two abstract methods that handle the measurement of elapsed clock cycles. The content of these functions is omitted here to keep the listing short. Each assembler instruction needs certain arguments. Some of them are register values and some of them constants which has to be annotated at the moment. Also the registers need to be allocated manually but we like to improve this in the feature with register allocation techniques borrowed from compiler research.

With the architecture description and the abstract assembly code it is possible to develop a code generator that creates the operating system code for a specific hardware platform. A simplified overview of the process is given with Figure 4. The

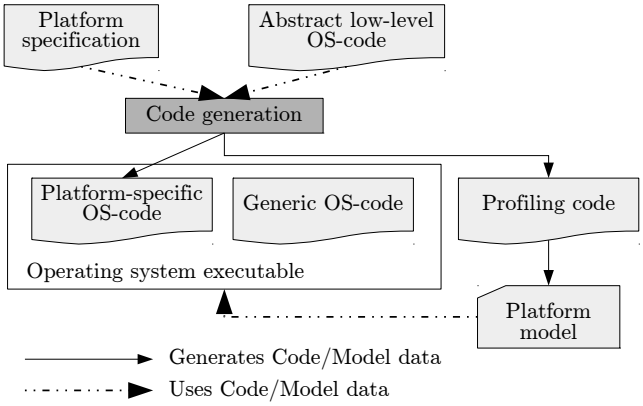


Fig. 4: Concept of operating system with abstract code

code generation combines one specific platform architecture with the abstract code and generates the assembler code for the architecture. This is then integrated with the generic program code of the operating system. The code generator can also be used to create comprehensive profiling code for the creation of a timing behavior description for the platform, that can also be used by the operating system as a base for optimizations like cache management to get a predictable system.

III. Evaluation

As a proof of concept we chose to develop a memory read performance profiler with the abstract language, because it is essential for the operating system to have information on the platforms memory performance to get predictable execution times. We want to use the generated information within our prototype operating system for the cache management [5].

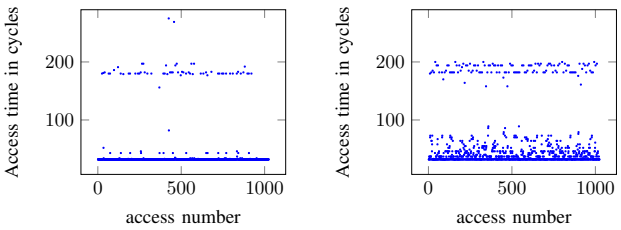
We implemented the presented languages with the Eclipse Modeling Framework (*EMF*) and *Xtext* [7] as this allows rapid prototyping of our domain specific languages and code generation which is helpful to quickly adapt the language to the changing demand as we developed our requirements to develop an operating system with abstract low-level code. We evaluated our implementation of the code generation with a *Samsung Exynos 4412* ARM-processor on a prototype operating system where no other load is simultaneously active.

The results of two generated benchmarks are shown on Figure 5. We evaluated two abstract benchmarks. One benchmark warms up the private cache of a processor by iterating over a memory range with the size of the private cache and finally iterates over the same range and measure the access times. The results are shown in Figure 5a. Another test is shown in Figure 5b where the main memory is tested without warming up so that we get many more cache misses.

Although the profiling code for now was only generated for an *ARM* processor, it is possible to adapt it to other processors in the future.

IV. Conclusion & Future Work

We demonstrated that it is possible to create abstract low-level code that can be transformed to architecture-specific



(a) L1-Cache profiling (b) RAM accesses

Fig. 5: Memory read benchmarks

assembly code by providing a simple architecture description. Although we could only present some preliminary results for now we intend to improve on this in the future.

One use case for the code generation process can be to write abstract profiling code once and then run it on many hardware platforms. To do so would require a good execution base to get reliable results. We intend to run profiling code on our prototype operating system [5]. But it would be interesting to see if it is possible to generate code that could be run on a operating system like Linux to get much better hardware support right away. A possible solution would be to generate Linux kernel modules that take control over the system and run the profiler code exclusively for a limited time. This would allow a broad range of hardware architectures to be analyzed. These models could be used by real-time operating systems to adapt them on specific hardware properties.

As hardware platforms are getting more difficult to develop for it would be handy to write low-level operating system code for hardware features like memory management, memory address translation and other things only once. To achieve this our abstract languages need to evolve to provide the necessary means.

A distant goal we would like to aim for is to generate an open source database with hardware models that describe the hardware in a way that is especially useful for the design and implementation of operating systems.

References

[1] J. M. Calandrino and J. H. Anderson, "Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study," in *20th Euromicro Conf. on Real-Time Sys. (ECRTS '08)*, Jul. 2008, pp. 299–308.

[2] D. Dasari, B. Akesson, V. Nelis, M. Awan, and S. Petters, "Identifying the sources of unpredictability in COTS-based multicore systems," in *08th IEEE Int. Symp. on Industrial Embedded Systems (SIES 2013)*, Jun. 2013, pp. 39–48.

[3] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *27th Int. Symp. on Comp. Arch. (ISCA '00)*. New York, NY, USA: ACM, 2000, pp. 128–138.

[4] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, "Making shared caches more predictable on multicore platforms," in *25th Euromicro Conf. on Real-Time Sys. (ECRTS '13)*. IEEE, Jul. 2013, pp. 157–167.

[5] H. Borghorst and O. Spinczyk, "Increasing the predictability of modern COTS hardware through cache-aware OS-design," in *11th W'shop on OS Platf. for Emb. Real-Time App. (OSPERT '15)*, Jul. 2015.

[6] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system," in *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, Sep. 2009, pp. 261–270.

[7] "Xtext," https://eclipse.org/Xtext/, accessed: 2016-05-23.