A HETEROGENEOUS PLATFORM DESCRIPTION MODEL FOR REAL-TIME SCHEDULING TOOLS

Duc Anh Nguyen
Abstract

A HETEROGENEOUS PLATFORM DESCRIPTION MODEL FOR REAL-TIME SCHEDULING TOOLS

Duc Anh Nguyen

Real-time system analysis tools and WCET analyzers both need information about the hardware to obtain the results. However, the current hardware models are either too abstract or too detailed. Therefore, this thesis proposes a new hardware description model that achieves the balance between two spectra; hence it is easy to use by the users without hardware expertise while not compromising too much accuracy. Along with the new model, an accompanying application is implemented to help the user work with the new model easier with its drag and drop interface. Furthermore, the application can function as a WCET analyzer by connecting to the Gem5 simulator, which facilitates simulating and measuring the execution time of the programs.

Handledare: Morteza Mohaqeqi
Ämnesgranskare: Wang Yi
Examinator: Johannes Borgström
IT 20 091
Tryckt av: Reprocentralen ITC
# Table of Contents

List of Figures .................................................................................................................. III

List of Acronyms ................................................................................................................ IV

1 Introduction and background .......................................................................................... 1
  1.1 Real-time systems .................................................................................................... 1
  1.2 Verification of the real-time systems ...................................................................... 1
  1.3 The worst case execution time of a software program ........................................... 3
    1.3.1 Static analysis .................................................................................................. 3
    1.3.2 Measurement ................................................................................................. 4
  1.4 Hardware model in analysis tools .......................................................................... 4
  1.5 Moving toward heterogeneous systems .................................................................. 6

2 Thesis goals and contributions ....................................................................................... 7
  2.1 Goals of the thesis .................................................................................................. 7
  2.2 Structure of the thesis ............................................................................................ 7

3 Related work .................................................................................................................. 8
  3.1 Hardware models .................................................................................................... 8
  3.2 Simulators ................................................................................................................ 8

4 Gem5 input hardware model .......................................................................................... 10

5 New hardware description model .................................................................................. 13
  5.1 Model specification ............................................................................................... 13
    5.1.1 Processor ....................................................................................................... 13
    5.1.2 Cache ............................................................................................................. 14
    5.1.3 Memory ......................................................................................................... 15
    5.1.4 Link ............................................................................................................... 16
    5.1.5 Group ............................................................................................................ 16
  5.2 Model JSON format ............................................................................................... 16
  5.3 Model example ....................................................................................................... 18

6 Application overview ..................................................................................................... 21
  6.1 Application functionalities ...................................................................................... 21
  6.2 How application works ........................................................................................... 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3</td>
<td>Gem5 simulation and measuring WCET</td>
<td>25</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Gem5 Python configuration file</td>
<td>25</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Translating model to arguments and simulating</td>
<td>26</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Extracting output statistics</td>
<td>28</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion and future work</td>
<td>30</td>
</tr>
<tr>
<td>7.1</td>
<td>Future work</td>
<td>30</td>
</tr>
<tr>
<td>7.1.1</td>
<td>The hardware description model</td>
<td>30</td>
</tr>
<tr>
<td>7.1.2</td>
<td>The application</td>
<td>30</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>Appendices</td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>A</td>
<td>Sample Gem5 Python configuration code</td>
<td>35</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Overview of an analysis tool</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>A more detailed overview of the analysis tool</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Phases of WCET computation, taken from [10]</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>A sample output of Gem5 simulator</td>
<td>5</td>
</tr>
<tr>
<td>1.5</td>
<td>Two purposes of the hardware model</td>
<td>5</td>
</tr>
<tr>
<td>4.1</td>
<td>Simple Gem5 UML class diagram</td>
<td>10</td>
</tr>
<tr>
<td>4.2</td>
<td>Accuracy comparison between different CPU and cache models</td>
<td>11</td>
</tr>
<tr>
<td>5.1</td>
<td>The simple diagram of the model</td>
<td>14</td>
</tr>
<tr>
<td>5.2</td>
<td>Model JSON file structure</td>
<td>17</td>
</tr>
<tr>
<td>5.3</td>
<td>Structure of the objects</td>
<td>17</td>
</tr>
<tr>
<td>5.4</td>
<td>Structure of the connection object</td>
<td>18</td>
</tr>
<tr>
<td>5.5</td>
<td>A traditional design containing two CPUs and a chipset</td>
<td>18</td>
</tr>
<tr>
<td>5.6</td>
<td>An abstract model of the architecture in figure 5.5</td>
<td>19</td>
</tr>
<tr>
<td>5.7</td>
<td>The Raspberry Pi 4 Model B</td>
<td>20</td>
</tr>
<tr>
<td>5.8</td>
<td>The Raspberry Pi 4 Model B described by our model</td>
<td>20</td>
</tr>
<tr>
<td>6.1</td>
<td>Components in the main UI</td>
<td>21</td>
</tr>
<tr>
<td>6.2</td>
<td>The toolbar and its functions</td>
<td>21</td>
</tr>
<tr>
<td>6.3</td>
<td>A platform with 4 components</td>
<td>22</td>
</tr>
<tr>
<td>6.4</td>
<td>Pseudo code of the model data structure</td>
<td>23</td>
</tr>
<tr>
<td>6.5</td>
<td>Extended JSON file generated by our application</td>
<td>24</td>
</tr>
<tr>
<td>6.6</td>
<td>Set the path to Gem5 project and the desired architecture</td>
<td>26</td>
</tr>
<tr>
<td>6.7</td>
<td>An example system model, the cache is configured in the processor property pane</td>
<td>27</td>
</tr>
<tr>
<td>6.8</td>
<td>A command line to invoke the Gem5 simulator</td>
<td>28</td>
</tr>
<tr>
<td>6.9</td>
<td>The simulation dialog in the application, showing the simulation result</td>
<td>29</td>
</tr>
</tbody>
</table>
# List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WCET</td>
<td>worst-case execution time</td>
</tr>
<tr>
<td>AUTOSAR</td>
<td>Automotive open system architecture</td>
</tr>
<tr>
<td>FSB</td>
<td>Front-side bus</td>
</tr>
<tr>
<td>KVM</td>
<td>Kernel-based virtual machine</td>
</tr>
<tr>
<td>MSHR</td>
<td>miss status holding register</td>
</tr>
<tr>
<td>AADL</td>
<td>Architecture Analysis and Design Language</td>
</tr>
</tbody>
</table>
1 Introduction and background

1.1 Real-time systems

Many critical computer systems in our society require real-time computing, where not only functionalities need to be correct, but also time constraints need to be preserved; otherwise, the result will have no value. Examples of systems that require real-time computing include air traffic control, self-driving car, nuclear power plant, avionic system, healthcare systems, virtual reality, and interactive systems. Real-time systems can be classified into two types: hard and soft real-time. When a soft real-time system misses a deadline, the value of the result may degrade over time or has no merit, but no deadly consequence happens. For instance, a video streaming application is a soft real-time system. The application has to fetch and process the data fast enough so that the viewers have a smooth experience, yet missing some deadlines will result in downgraded video quality. On the other hand, a hard real-time system must meet its deadlines. If not, a catastrophic consequence can happen [1]. Take a pacemaker implanted inside the patient’s body as an example. If a pacemaker fails to operate or respond in time, the patient can be dead. Therefore, these systems must be validated thoroughly before going into use.

To ensure that all the tasks meet their deadlines, the task execution must be coordinated by the operating system (OS) scheduler. The OS schedulers use a variety of scheduling algorithms such as EDF, RM, or DM [2]. To know whether the algorithm can schedule the task set, the system must go through the verification step.

1.2 Verification of the real-time systems

Verification is a crucial step in the software development process. It is even more important for a real-time system because missing a deadline results in severe outcomes. There are two primary approaches to verify the timing properties of a real-time system: theoretical analysis and simulation. Some of the methods employed by the theoretical analysis approach include schedulability analysis, model checking, and state-space exploration, etc. The task source code is modeled with different methods, ranging from a simple model such as Liu and Layland [3] to complicated graphs representing the internal structure of the program [4]. Although a comprehensive model checking method can verify that the program is free of bugs, the analysis methods that take into
account all possible interactions between software and hardware components have the scalability issue [5].

The simulation approach verifies the timing properties by directly running the task set on the virtual platform and sees whether there is any task that misses a deadline. The exactness of the simulation depends on the simulator. However, there is a trade-off between the precision and speed of the simulation [6].

The analysis tools usually combine both approaches, as in figure 1.1. The software model is passed through the tool, and the output specifies if all the timing constraints are satisfied.

![Figure 1.1: Overview of an analysis tool](image1.png)

Besides the software, the hardware platform the tasks execute on also plays a big role in deciding the schedulability of the task set. The number of processing units determines how many tasks can be executed in parallel, the cache introduces non-determinism into special memory access, the interconnection topology affects the communication latency, and so on. It is important for the analysis tool to take the characteristic of the hardware into consideration, as shown in figure 1.2.

![Figure 1.2: A more detailed overview of the analysis tool](image2.png)
1.3 The worst case execution time of a software program

One of the essential properties describing the timing properties of the software program is the worst-case execution time (WCET). The WCET is the total amount of time a processing unit needs to execute the program code. The WCET is present in all the scheduling algorithms research. For a simple workload model, Liu and Layland’s model [3], the WCET is one of the two properties describing the task, besides the period. Thus, deriving the WCET of the provided software program is a crucial step before schedulability analysis and verifying the task.

The problem of obtaining a strict WCET is reducible from the halting problem [7], which is generally unsolvable. The WCET of a task depends on the input, the current state of the system, which can have an infinite number of cases. However, the software, written for the real-time system, often follows a set of restricted rules (e.g., The Power of 10 [8]) which ensure the program has no recursion, no infinite loop, bounded iteration, and terminating after an amount of time. Similar to verifying a real-time system, there are two different ways to derive the WCET of a given program: static analysis and measurement [7]. Subsequence sections will go deeper into each approach.

1.3.1 Static analysis

This method does not execute the code directly. It takes the code and an abstract model of the underlying hardware then calculates the upper bound of the WCET. The source code, which may have some supporting annotation, is compiled down to binary or assembly format. From it, the code is segmented into basic blocks (a basic block is a block of code with only one execution flow). The execution time of a basic block can be obtained by adding up each instruction’s cycles, which can be found in the CPU technical reference. Then, the basic blocks are connected to form a control flow graph. From it, we identify the path with the longest execution time. That is the result we need [9].

Unfortunately, the computer architecture is getting more and more complex, with caches introducing non-determinism in memory access latency (cache hit/miss), multicore and multithreaded execution introducing more types of resource contention and timing anomaly. The static analysis approach becomes very difficult, as too many factors affect the execution. Nowadays, the static analysis tools comprise of many modules, each analyzes different aspects of execution. For instance, aiT [10] is a commercial WCET analyzer. aiT examines the compiled source code, with some supporting annotations, in many phases: value analysis, loop bound analysis, cache analysis, pipeline analysis, and path analysis. Figure 1.3 visualises the execution steps of aiT.
However, the aiT tool only supports the system with CPUs. As GPU computing and the use of accelerators for higher performance come into play, the system architecture is getting even more complex. Achieving a tight upper bound using static analysis becomes extremely difficult and sometimes impossible.

### 1.3.2 Measurement

The measurement-based method executes the program on the target hardware and measures the execution time. Although testing on real hardware gives the best result, the virtual platform simulated by a simulator facilitates earlier testing, lower the cost, and explores the design space easily. Moreover, testing some programs in real conditions (e.g., airplane, satellite) is not possible. There are many ways to perform the measurement. Instrumentation code that collects timestamps or CPU cycle can be added to the start and end of the program then we calculate the difference. The output of the simulator can be collected and processed to find out the execution time. This way is more complex, but the result is better since no modification is made to the measured program. For example, figure 1.4 shows a sample output of the Gem5 simulator [11] when running a hello world program on a simple platform. On the last line is the number of simulated ticks, which is one possible value for the WCET.

The measurement method can only test a subset of the input domain and derive the WCET for that subset. However, it will be the true upper bound if the subset contains the worst-case input, or if the worst-case input is known, only one execution is enough. Nonetheless, finding the worst-case input is hard on complex systems and task sets.

The result of the measurement-based method can be used to be cross-validated with the static analysis method. The value produced by the two methods should not differ too much; otherwise, it can be an indication of some incorrect configuration.

### 1.4 Hardware model in analysis tools

From section 1.2 and 1.3, it is seen that the information of the underlying hardware is a necessary input data to the analyzers. Therefore, before analyzing the system, one
needs to determine a hardware model that captures the essential characteristics of the hardware platform.

Nowadays, there are many hardware description tools. Verilog [12] and VHDL [13] both can describe the hardware chip on the logic gate level. With them, we can have a very detailed structure and behavior of an integrated circuit. However, to use these tools, users are required to have a broad knowledge of logic, electronic design; and master the tools to understand and make changes when needed. Additionally, not every small detail has a large effect on the timing properties of the software, leading to time-consuming and inefficient analysis. Therefore, the hardware model must avoid being too detailed.

On the other hand, the hardware model should not be too abstract. A model created according to AUTOSAR hardware model specification [14], for example, can document the high-level structure of the system, with the processor, connector, port, etc. But
performing analysis based on the information extracted from that model will lead to inaccurate results. Hence, the most appropriate hardware description model must achieve a balance between abstraction and detail or efficiency and accuracy of the analysis result.

1.5 Moving toward heterogeneous systems

Nowadays, real-time systems are moving toward heterogeneous platform due to its high performance. Some example systems in [15] and [16] use an ARM processor combined with an FPGA-based accelerator to improve the performance of the compute-intensive task. Performing analysis on these platforms poses a big challenge since the affecting factors now are not just processor, memory, and cache but also the bus, interconnection devices, the GPU, and the accelerator. The accelerator, by nature, is developed for a specific task, so its internal components are different from one to one. This raises the need for a new hardware model with the capability to describe a platform with multiple types of processing units, diverse interconnection topology.
2 Thesis goals and contributions

2.1 Goals of the thesis

This thesis aims to create a new hardware description model that overcomes the issues stated in the previous sections. Those are:

- Avoid being too abstract so that the analysis results have reasonable accuracy.
- Avoid being too detailed so that the analysis would not take too much time, and non-expert users can also use the model.
- Be able to describe the platform with many types of computing units (CPU, GPU, accelerator) and arbitrary interconnection topology.

And the resulting model will be used for two purposes:

- Providing the hardware information to the analysis tools that analyze the timing properties of the real-time system.
- Providing the hardware information to the WCET analyzer, thus the output execution time is not too pessimistic.

Besides the new model, we also present a new application that is capable of:

- Create and edit the new proposed model using a graphical interface.
- Given the program code, the application invokes a simulator to simulate the program on the modeled hardware, then extracts the execution time.

2.2 Structure of the thesis

This thesis is structured as follows: Section 1 is an introduction and provides the background of the problem, section 2 states the goals and contributions of the thesis, section 3 reviews the related work, section 4 briefly goes through the Gem5 input model, section 5 demonstrates the new proposed model and the parameters derived, section 6 shows the application and how it works, section 7 discusses future development and concludes the thesis.
3 Related work

3.1 Hardware models

There are several languages and standards describing the hardware system, from abstract description to detailed models used in the simulators. The ECU resource template specification [14] published and maintained by Automotive open system architecture (AUTOSAR) [17] models the hardware in a very abstract sense. From it, we can see a general topology of the system, but we cannot derive any useful parameters from it, as it is very abstract. Depending on the user’s requirement, it can be extended by defining additional properties or classes. Another specification [18] from the HSA Foundation [19] specifies comprehensive requirements that a system must meet to support a wide assortment of data-parallel and task-parallel programming models. Since the heterogeneous system is what this specification aims for, it serves as an important guideline of how a system looks like from the software point of view. The Architecture Analysis and Design Language (AADL) [20] is an architecture description language standardized by SAE [21]. AADL focuses on modeling both software and hardware of the embedded real-time systems. The model of a system serves as design documentation, functional and timing verification, and code generation. However, all the properties, including WCET of the provided code, communication latency, and memory access latency, is inputted manually and is assumed to be known beforehand. Nonetheless, how the language is structured, its components and definitions greatly influence the design of our model.

3.2 Simulators

On the hardware simulator side, there are many tools available with different goals, performance, and development activities. Simics [22] is a full system simulator that is capable of running unmodified software on a virtual platform. Simics supports a wide range of architectures (e.g., Alpha, ARM, x86). However, Simics aims to simulate the correct functionalities of the target software in a fast and efficient way by abstracting the timing aspect of the physical hardware, thus sacrificing timing correctness. SimpleScalar [23] is an open-source architecture simulation with a wide range of supported architectures and types of CPU, from simple unpipelined processors to pipelined out-of-order CPU. Unfortunately, SimpleScalar now is no longer developed and maintained. Qemu [24] is another open-source machine emulator and virtualizer. It emulates CPUs through
dynamic binary translation so that it achieves impressive performance. Additionally, Qemu can combine with Kernel-based virtual machine (KVM) to simulate with near-native speed. Finally, Gem5 [11] is a very popular computer architecture simulator in both academia and industry. Gem5 can simulate different types of CPU, a handful of cache coherent protocols, and arbitrary topology. Additionally, Gem5 now is still under heavy development and an active support team.

In [25], the performance of Gem5 is evaluated and compared with the real hardware execution. Depending on the type of software, the mismatch between Gem5 simulation and real hardware execution ranges from 0.47% to 17.94%. This is due to Gem5 does not model the timing behavior of DDR memory correctly. In [6], the performance between Gem5 and Qemu is compared. The result of which is better depends on the chosen benchmark. But overall, Gem5’s ARM out-of-order CPU gives the execution time closed to the real-life value. The highly accurate simulation of Gem5 is one of the reasons that we choose it.
4 Gem5 input hardware model

One of the functionalities of our application is to obtain the WCET of programs by using a simulator. We choose the Gem5 simulator for this purpose because of the wide range of supporting CPU architectures, the ability to model comprehensive cache structure, and partial support for GPU. But the most important is, Gem5 is a cycle-accurate simulator. A cycle-accurate simulator simulates the operation with the clock cycle granularity, as opposed to the functional-accurate simulator, which focuses on replicating correct functionalities, but the timing aspect is not guaranteed. Therefore, Gem5 gives a good estimation of the execution time of the programs. In this section, we will go through the way Gem5 structures its simulated hardware configuration.

Gem5 is implemented in C++ and Python and follows the object-oriented paradigm. In figure 4.1 is the simple UML class diagram of the most basic and important classes. Each branch contains the basic component of a computer: CPU, cache, main memory, and interconnect components. Below we will go deeper into each component.

Gem5 provides several CPU models, each for different purposes. They are:

- AtomicSimpleCPU: The simplest CPU model. It is a purely functional, in-order CPU, using atomic memory access, which means the memory access does not take resource contention, queuing delay into account.

- TimingSimpleCPU: Like AtomicSimpleCPU, TimingSimpleCPU is also purely functional. The difference is, it uses timing memory access, trying to model realistic timing by considering resource contention and queuing delay.
- MinorCPU: An in-order pipeline processor model, with timing memory access.
- DerivO3CPU: An out-of-order pipeline CPU model. This model is made with the best effort from the Gem5 team to achieve high timing accuracy by actually executing the instructions at each execution stage of the pipeline.

To measure execution time, we choose the DerivO3CPU model. As in [25] and [6], the O3CPU model exhibits the timing property closest to the real hardware. MinorCPU, an in-order pipeline CPU, can be used if the system uses an in-order CPU.

Gem5 comes with two types of cache implementation, the classic cache, and the Ruby cache. The reason is Gem5 is a combination of m5 tool from Michigan and GEMS from Wisconsin. The Ruby cache comes from GEMS, whereas the m5 uses the classic one. The classic cache, in which the Cache class and BaseCache class is a part of, implements a simplified and inflexible MOESI cache coherent protocol. The Ruby cache, on the other hand, is designed to model cache coherence in detail, with its language to define coherence protocol, and Garnet [26] to modeling fine-grained interconnection network. Ruby cache supports a large range of coherence protocols such as MI, MSI, MOESI.

![Accuracy comparison between different CPU and cache models](image)

Figure 4.2: Accuracy comparison between different CPU and cache models

CPU and cache in Gem5 are modeled in many levels of accuracy and efficiency, therefore choosing and configuring the CPUs and caches has a large effect on the simulation result. From figure 4.2, we can see that using the O3 CPU model with Ruby Garnet cache gives the best accuracy. However, configuring the Ruby cache network is a work-intensive task, requiring writing a state machine in the language provided by Ruby and recompile Gem5 if we make changes to the protocol. Moreover, [6] shows that using the classic cache also gives a good result.
The DRAMInterface contains many properties of a typical DRAM, including electric current in the physical operation, the latency of row, column access, precharge, refresh, etc. It has so many small details that make implementing a DRAM out of this interface require in-depth knowledge about the DRAM. Fortunately, Gem5 also provides some implementation class based on the commercial DRAMs, for instance: Micron DDR3-1600-8GB, Micron DDR3-2133-8GB, SK Hynix GDDR5-4000-8GB.

In the hardware system, there are many ways to connect CPUs, caches, and memories [27]. In Gem5, CoherentXBar class models the crossbar topology. Bridge topology is also supported. These classes are used in the system with the classic cache. In the system with Ruby cache, more topologies are supported but they require different implementation.

Gem5 can run in two different modes called syscall emulation (SE) or full system (FS). SE mode does not attempt to simulate the entire hardware and operating system, instead, it emulates the Linux system call. When the simulated program makes a system call, Gem5 takes it, doing necessary operations, and returns the result to the caller. Whereas, in FS mode, Gem5 emulates the entire hardware system, allowing an unmodified operating system (OS) to run on it. This is similar to running a virtual machine. In comparison, SE mode is much faster and easier to configure because in FS mode Gem5 has to set up devices, BIOS, boot up kernel and OS, and this process could take a significant long time, up to a few hours. In contrast, if we need extremely high fidelity in modeling for some very strict real-time systems, and the effect of OS interaction and noise cannot be ignored, FS mode is the right way to go.

A sample configuration is provided in figure A.1 in the appendix. This is a simple system with a simplified single-core CPU model, running at 1GHz, does not have a cache, is connected to a DDR3, 8GB RAM, and running a hello world program. The program is simulated in the Syscall Emulation (SE) and the output is shown in figure 1.4. The simulation statistics are written to a file called stats . txt, which contain hundreds of useful statistics. This is an important source of data to obtain the execution time of the program.
5 New hardware description model

In this section, we will illustrate our new model for hardware description and show some examples.

5.1 Model specification

Our new model has the following components:

- Processor
- Cache
- Memory
- Link
- Group

Except for Group, which is just syntactic support to the model, every component above is the vital part of a computer. All of the components can be seen as black boxes with predefined properties describing the internal characteristics. Each of them can connect to the others, regardless of the type. This flexibility facilitates modeling arbitrary system topology. Indeed, not every arbitrary system is practical or exists in real-life. Verifying whether the model is workable or exists in real-life is the job of the program that uses the model. Different programs can interpret and verify the model differently. For instance, we use Gem5 to measure the WCET of tasks, so in the code that connects Gem5 to the model, we write the logic that reads the model, checks if it can be transformed into the Gem5 input, then do the actual translation.

The diagram in figure 5.1 shows the properties and relationship between model components. In the next few sections, we will present each component in detail.

5.1.1 Processor

A processor model represents a unit consuming workload from a task. Generally, this model component should stand for all kinds of computing units, such as CPU, GPU, accelerator. However, right now we mostly consider the CPU and its properties because of its prevalence in all the embedded systems. GPU, on the other hand, is not widespread
in the field of embedded devices. It usually shows its presence in high-performance computing. A list of processor’s properties includes:

- number of cores: each core will be the same as one another
- frequency: clock frequency of the CPU
- cache hierarchy: the cache structure of this processor.

A processor with many cores and 2 or 3 cache levels can be specified easily. If more fine-grained control is needed (e.g., asymmetric processor) or testing with new kind of processor architectures, a processor can act as a single computing core by setting the number of cores to one and specifying zero cache hierarchy. With caches and links (defined in the next sections) any kind of processor architecture can be represented.

### 5.1.2 Cache

Cache represents intermediate memory storage which stands between the computing units and the main memory such as RAM. They exploit the temporal and spatial locality characteristics of the program to speed up the memory access latency. However, the cache is smaller than the main memory, and different cores can have different instances of cache, holding copies of the same memory block. So cache coherency protocol is needed to keep the memory consistent. Due to its non-deterministic nature, the cache makes real-time schedulability analysis more difficult. The cache has the following properties:

- cache size: total size of the cache, in KB
- cache line size: size of a cache line, in KB
• associativity
• inclusion policy: include, exclude or NINE (neither include nor exclude)
• replacement policy
• tag latency: cycles needed to lookup a tag in the cache
• data latency: cycles needed to access the data (cache hit)
• response latency: delay on the return path on a cache miss
• mshrs: the number of miss status holding register (MSHR). This is the number of outstanding requests a cache can handle at a single time. This register enables memory-level parallelism. However, miss-use of MSHR can severely affect the program performance [28]

5.1.3 Memory

The memory model serves as DRAM and it has the following properties:
• DRAM chip/device size
• number of chip per rank
• number of rank
• number of channel
• number of bank per rank
• chip bus width
• maximum latency

The first three properties are to calculate the capacity of a DRAM. DRAM stores data in a unit so-called rank, each rank contains several chips or devices [29]. Memory requests can be served in parallel between multiple chips. So total capacity of a DRAM is calculated by

\[
memory = chip\_size \times number\_of\_chip\_per\_rank \times number\_of\_rank
\]

The number of channels can be inferred from the DRAM name. For instance, DDR3 means three channels, DDR4 means four channels. The maximum latency property is an abstraction over the sophisticated DRAM latency. DRAM can exhibit different latency depends on the memory access pattern and comprise of many small latencies such as RAS to CAS delay, CAS latency, row precharge time, active to precharge delay. Choosing the maximum delay time possible for every memory access could be pessimistic, but it is a safe upper bound and avoids unnecessary detail.

© Uppsala University

Duc Anh Nguyen
The memory model can also represent persistent storage such as hard disk and floppy disk. However, usually, the entire program is loaded into the main memory (RAM) after some warm-up time, except for an enormous source code or the program that relies on persistent storage like a database or data streaming application. Therefore, they are not considered now.

5.1.4 Link

A link can be any kind of communication method between 2 elements of any kind. It can be a bus in the CPU, a point-to-point connection between 2 cores, a PCI-e to connect to other hardware, a network switch, and so on [27]. Note that this model entity not only models connection inside a system but also models network between physically separated machines. The properties of the link are:

- type: bus, point-to-point (AMD’s HyperTransport, Intel’s QuickPath Interconnect, etc), PCI-e bus, network switch,...
- protocol: CAN
- bandwidth
- maximum latency

5.1.5 Group

A group only serves the purpose of organizing the above components into modular, reusable sets. Grouping components together does not imply any specific configuration in the physical machine.

5.2 Model JSON format

To represent the model, we have two different methods: graphical visualization, used in our application (section 6) and JSON format. JSON format gives the best balance between human-readable and machine-readable representation, allowing both manual editing and machine generation to easily be made. A JSON file containing our model has the following form shown in figure 5.2.

As in figure 5.2, the "objects" array contains the definition of all model objects in the system, and the "connections" object holds the information of which object connects to which. The structure of a model object is shown in figure 5.3 and the connections in figure 5.4.
As illustrated in the figure, each object has:

- An ID: we use UUIDv4 to guarantee uniqueness between objects.
- A type: the type name in string.
- A params object: this object holds the properties of the model listed in the section 5.1.

The connections object is structured like an adjacency list, an ID of an object as a key, and the value is a list of IDs of all objects connected to the object which has the ID in the key.

Using JSON format and the above structure have several advantages. One of them is the JSON format is easily modified by both human and machine, as stated. Another advantage is that it is easy to extend the structure. New tools or standards can add new key-value pairs along with the existing ones. If the old key-value pairs mentioned above still exist, then the file is still readable by the old tools.
5.3 Model example

Here we will show some example systems and model them using our proposed model above.

Example 1  We start with a simple traditional design: a CPU with two chipsets, a Northbridge and a Southbridge.

Figure 5.5: A traditional design containing two CPUs and a chipset

Here is a common CPU structure that can be found on most old CPUs [29]. Two CPU cores are connecting to a Northbridge using the common Front-side bus (FSB). Only one core can use the bus at a time. The RAM and the memory controller are attached to the Northbridge. The non-volatile memory (e.g., hard disk, USB) or other components connected by the PCI-E bus must go through the Southbridge. The Northbridge connects to the Southbridge by an internal bus (Intel calls it Direct Media Interface, while AMD
calls it Unified Media Interface). To model this architecture, we can use a processor with two cores, a memory to represent RAM, and the link entities for FSB, Northbridge, Southbridge, internal bus, and PCI-e bus. The abstraction model of the architecture in figure 5.5 is shown in figure 5.6.

![Diagram of architecture](image)

**Figure 5.6: an abstract model of the architecture in figure 5.5**

**Example 2** For this example, we will use real-life embedded hardware, which is the Raspberry Pi 4 Model B. The full technical detail can be found on [30], here we will give a short technical spec of the devices

- CPU: Quad core Cortex-A72 (ARM v8) 64-bit SoC 1.5GHz, with 48KB instruction cache, 32KB data cache for each core, and 512KB shared level 2 cache [31]
- 4GB LPDDR4-3200 SDRAM
- Other features such as Bluetooth, USB ports, wireless, ethernet.

In figure 5.8, we demonstrate the description model of the Raspberry Pi 4 device. Here we left out the IO and the communication components since our model does not support them yet.
Figure 5.7: The Raspberry Pi 4 Model B

Figure 5.8: The Raspberry Pi 4 Model B described by our model
6 Application overview

6.1 Application functionalities

As stated in the introduction section, another goal of the thesis is to create an application that makes creating the hardware model easy and fast. In figure 6.1, we present the application we have developed so far.

![Components in the main UI](image)

**Figure 6.1: Components in the main UI**

![The toolbar and its functions](image)

**Figure 6.2: The toolbar and its functions**

The main UI of the application consists of a toolbar on the top, a tab list, a list of entries on the left, a list of selected entry’s properties on the right, and the middle working space. The toolbar, shown in figure 6.2, contains frequently used operations, including file operations (new, save, open), model creation, and simulation related features. The middle space is a drag-and-drop canvas so that we can drag to change the position of the entry and connect different entries to form a system topology. The users can save
their work for later use with the save and load file feature of the application. The model is written to the file in the JSON format, following the syntax described in section 5.2, with additional key-value data that is specific to the internal working of the application.

For example, figure 6.3 illustrates a platform with a processor connecting to a cache, a link which represents a bus, and a memory, which is DRAM. The memory component is selected, so on the right is the list of properties describing the memory. As shown in the figure, this memory represents a DRAM with 4GB and 10ns maximum memory access latency. Users can look at the left panel for a list of current objects and can click to select other objects for inspection.

![Figure 6.3: A platform with 4 components](image)

### 6.2 How application works

The application is built using the following frameworks and libraries:

- **JavaFX**: JavaFX is the GUI library for Java desktop applications. It is a newer standard and a replacement for the old Swing library.

- **JFoenix**: It is the GUI library, based on JavaFX, that provides material-UI style components.

- **Spring framework**: Spring is the biggest and most popular framework for Java applications. Although the framework is tailored toward server application, we utilize its top-notch support for dependency injection and an event system to make our source code modular and decoupled.

- **Gradle**: Gradle is the state-of-the-art dependency management and a build tool for Java application that makes including a dependency or running the application
Gradle automatically resolves dependencies and builds the application. The output can be an executable JAR or a standalone runtime image.

The UI is separated into small parts. Those are the toolbar, the tab list, the main canvas, the left, and right pane. Each of them has a corresponding controller class. All the states of the model and the application is held in one central class. Thanks to the dependency injection and event system, all the classes are loosely coupled, and new features can be added at ease.

To store the model, we make use of the adjacency list data structure, with a list of model objects, similar to the nodes in the graph, and the list storing the connections or the edges in the graph. This structure conforms to the JSON structure already presented in section 5.2, thus making serialization and deserialization between the state of the program and the JSON file easier. Besides the usual properties of the model in the JSON file, we add some application-specific data. The extended JSON object is presented in figure 6.5. The simple pseudo-code for the internal data structure is shown in figure 6.4.

```
List<ModelEntity> // holding the list of nodes in the graph
Map<EntityId, List<EntityId>> // the adjacent list, each entity on the left is connected to the list of entities on the right
```

Figure 6.4: Pseudo code of the model data structure
Figure 6.5: Extended JSON file generated by our application
6.3 Gem5 simulation and measuring WCET

Apart from creating and editing the model of a system, our application can translate the model into the form suitable for Gem5 and start a simulation to determine the execution time. First, we will demonstrate how we invoke Gem5 from our application.

6.3.1 Gem5 Python configuration file

Gem5 uses Python files to configure the system, assemble the components together, bind the workload to the processor, and start the simulation. Generating Python files from our Java application would be very hard. Fortunately, the Gem5 codebase provides a sample Python file that accepts command-line arguments to construct the system and simulate in SE mode. Therefore, first, we read the current system model, generating command-line arguments, then invoke the python script with the arguments as if we call it from the bash shell. The list of the arguments we use is shown in table 6.1.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--num-cpus</td>
<td>1</td>
<td>The number of CPU cores to simulate</td>
</tr>
<tr>
<td>--cpu-type</td>
<td>AtomicSimpleCPU</td>
<td>The type of CPU to run with</td>
</tr>
<tr>
<td>--cpu-clock</td>
<td>2GHz</td>
<td>CPU clock rate, all core will run at this frequency</td>
</tr>
<tr>
<td>--sys-clock</td>
<td>1GHz</td>
<td>System frequency, used by all other components</td>
</tr>
<tr>
<td>--mem-type</td>
<td>DDR3_1600_8x8</td>
<td>The type of DRAM to use</td>
</tr>
<tr>
<td>--mem-channels</td>
<td>1</td>
<td>The number of DRAM channel</td>
</tr>
<tr>
<td>--mem-ranks</td>
<td>None</td>
<td>The number of DRAM rank</td>
</tr>
<tr>
<td>--mem-size</td>
<td>512MB</td>
<td>Physical memory size of a single chip</td>
</tr>
<tr>
<td>--mem-channels-intlv</td>
<td>0</td>
<td>Memory channel interleave</td>
</tr>
<tr>
<td>--caches</td>
<td></td>
<td>Enable private level 1 caches for all cores</td>
</tr>
<tr>
<td>--l2cache</td>
<td></td>
<td>Enable level 2 cache shared by all cores</td>
</tr>
</tbody>
</table>
### Application overview

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--l1d_size</td>
<td>64kB</td>
<td>Size of level 1 data cache</td>
</tr>
<tr>
<td>--l1i_size</td>
<td>32kB</td>
<td>Size of level 1 instruction cache</td>
</tr>
<tr>
<td>--l2_size</td>
<td>2MB</td>
<td>Size of level 2 cache</td>
</tr>
<tr>
<td>--l1d_assoc</td>
<td>2</td>
<td>Level 1 data cache associativity</td>
</tr>
<tr>
<td>--l1i_assoc</td>
<td>2</td>
<td>Level 1 instruction cache associativity</td>
</tr>
<tr>
<td>--l2_assoc</td>
<td>8</td>
<td>Level 2 cache associativity</td>
</tr>
<tr>
<td>--cacheline_size</td>
<td>64kB</td>
<td>Size of a cache line</td>
</tr>
<tr>
<td>--cmd</td>
<td></td>
<td>The path to the binary workload</td>
</tr>
</tbody>
</table>

#### 6.3.2 Translating model to arguments and simulating

Before we start simulating, we set the path to the Gem5 codebase and determine the simulated architecture in our application setting, as illustrated in figure 6.6.

![Setting](image)

Figure 6.6: Set the path to Gem5 project and the desired architecture

Then the application reads the current system model, performs validation and translation in one go. Validation here does not verify the system is valid in real-life but rather verifies the model can be translated to the Gem5 command-line arguments in table 6.1 or not. The steps which the application carries out consist of:
• Find all processor objects. Make sure that all the objects have the same clock frequency, and count all the core and pass the result to the --num-cpu argument.

• Check if level 1 cache is present. If it does, it must be private to each core and have the same property values.

• Check if level 2 cache is present. If it does, it must be shared by all core.

• Find the memory object. There must be only one object.

• The caches, if present, must stand between the processor and the memory.

• The number of paths to the executable must be equal to the number of cores.

If the system model passes all the above steps, the equivalent Gem5 command line is created. For example, figure 6.7 demonstrates a simple system that is translatable to the command line, the equivalent command line is shown in figure 6.8.

![Figure 6.7: An example system model, the cache is configured in the processor property pane](image-url)
Here in this command line, we have to use the TimingSimpleCPU, although DerivO3CPU gives better accuracy. This is because of a bug in the Gem5 source code that makes simulating an x86 system with more than two DerivO3CPU result in an error.

### 6.3.3 Extracting output statistics

When the simulation finishes, Gem5 writes the output statistics to the files in the m5out directory. There are many useful stats tracked by Gem5, but for the measuring execution time purpose, we only find the numCycle stats for each core. The numCycle captures the number of simulated cycles for the corresponding core. Using the following simple formula, we can calculate the execution time of that core.

\[
\text{execution time} = \frac{\text{numCycle}}{\text{core’s frequency}}
\]

The result of the above command line is shown in figure 6.9. As we can see, each core executes a simple hello world program. The program prints "hello world" a fixed number
of times using a loop then exits. The first core executes the program in 850.679ns, while the second core takes 859.812ns to finish.

![Gem5 Simulation](image)

**Figure 6.9:** The simulation dialog in the application, showing the simulation result
7 Conclusion and future work

In this thesis, I have proposed a new hardware description model serving two purposes: providing hardware information (1) to the real-time system verification tools for timing analysis, and (2) to the WCET analyzer, which computes the execution time. The model is balanced between being too abstract, which produces an unrealistic result, and too detailed, which is inefficient and time-consuming. We also develop an application providing a graphical interface to create and edit the model easier. The application can be connected to the Gem5 simulator to measure the execution time of the programs.

7.1 Future work

This work can be extended in different directions. Below I will outline some possible research directions for each part of the thesis.

7.1.1 The hardware description model

Modeling the IO, sensor or actuator devices  A large amount of real-time embedded software operates on a stream of data generated from some sensor or the internet, and the output data may be pushed back to the internet or consumed in an actuator. One can imagine those as data source and sink. Modeling them is something we look forward to in the future.

Support for GPU, Accelerator  A heterogeneous system not only consists of asymmetric CPUs but also GPUs and accelerators as well. But the fact is that CPU, GPU, and accelerator can have vastly different architecture so that new model objects are needed. On the simulator side, Gem5 has one GPU model based on the AMD’s Graphics Core Next 3 (GCN3) microarchitecture, running on the Radeon Open Compute Platform (ROCM). Right now, it only works in SE mode. The support for FS mode, which is capable of running unmodified software, is currently under development. On the other hand, Gem5 does not support accelerator, but there are some other research groups has been developing a fork of Gem5 with support for accelerator [32], which is very promising to look at.

7.1.2 The application

Better support for Gem5  The current Gem5 support demonstrated in section 6.3 is very limited because of the inflexibility of the Python config file. To allow more types
of system model to be simulated on Gem5, a new way to translate is needed. The most promising method is to have a Python file that reads the JSON file and constructs the system from it. Because our model is already represented in JSON format, this seems to be an appropriate way.

**Better UI/UX**  The current UI/UX does not give a smooth experience in some actions, especially when editing the binary path in the processor. Improving UI/UX is one of our goals to make the application more user friendly.
References


Appendices

A Sample Gem5 Python configuration code
Program A.1: Sample code of the simple Gem5 system. I left out unnecessary setup code, leaving only code which setup and connect core component.

```python
# create the system we are going to simulate
system = System()

# Set the clock frequency of the system
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '1GHz'
system.clk_domain.voltage_domain = VoltageDomain()

# Create a simple CPU
system.cpu = TimingSimpleCPU()

# Create a memory bus, a system crossbar, in this case
system.membus = SystemXBar()

# Hook the CPU ports up to the membus
system.cpu.icache_port = system.membus.slave
system.cpu.dcache_port = system.membus.slave

# create the interrupt controller for the CPU
system.cpu.createInterruptController()

system.cpu.interrupts[0].pio = system.membus.master
system.cpu.interrupts[0].int_master = system.membus.slave
system.cpu.interrupts[0].int_slave = system.membus.master

# Create a DDR3 memory controller and connect it to the membus
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.master

# Connect the system up to the membus
system.system_port = system.membus.slave

# Create a process for a simple "Hello World" application
process = Process()

# Set the command
process.cmd = ['tests/test-progs/hello/bin/x86/linux/hello']

# Set the cpu to use the process as its workload
system.cpu.workload = process
system.cpu.createThreads()

root = Root(full_system=False, system=system)
m5.instantiate()
exit_event = m5.simulate()
```

© Uppsala University

Duc Anh Nguyen