Modeling of hardware software performance of high-tech systems

Peter van den Bosch
Océ Technologies B.V.
Venlo, The Netherlands
peter.vandenbosch@oce.com

Marcel Verhoef
Chess B.V.
Haarlem, The Netherlands
Marcel.Verhoef@chess.nl

Gerrit Muller
Embedded Systems Institute
Eindhoven, The Netherlands
Gerrit.Muller@esi.nl

Oana Florescu
Technical University Eindhoven
Eindhoven, The Netherlands
O.Florescu@tue.nl

This work has been carried out as part of the Boderc project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Dutch Ministry of Economic Affairs under the Senter TS program

Abstract

The performance of the control system is an important aspect of a machine. It would be a waste if a high-tech machine has been build such that it can physically achieve a high throughput, for example printed sheets of paper, but is limited because the software controlling it cannot keep up. Unfortunately, with current techniques it is hard to "predict" beforehand what the performance of the software will be when it finally runs in the real system on the real processor(s). There are two (extreme) ways to deal with it:

1. Over-dimension the hardware platform to make sure the software will run.
2. Implement the software, then run and evaluate its performance on the target hardware platform. Then use this information in the next design cycle.

The disadvantages of both approaches are clear. In the first situation the cost price of the entire system will surely be higher than necessary. In the second case, the design time is increased dramatically because more design cycles are needed. Therefore, it is important to strive to a development method that leads to fast design cycles for software performance, while having an accurate enough prediction. In this paper we will discuss a pragmatic modeling approach to design for performance in the domain of software intensive systems.

Problem formulation

As explained, the goal is to find or develop methods, techniques and tools that make it possible to predict the performance of software accurately based on only a small model that does not need a lot work to come up with. Obviously, there is a tension between the accuracy of the performance prediction and the amount of work needed to make the model. In general, it is even likely (but not proven) that it will require more work to make an *exact* prediction of the performance than it would be to create the whole system, run it and see how it performs.

During the process of performance modeling, and also during other Boderc activities, we realized that the goal of creating a model is not only to do an analysis and to make a prediction. Probably more important is the understanding that is obtained by creating the model, see also (Kostelijk 2005). This understanding leads to the ability to make better design choices and to be

able to understand the influences faster, thus decreasing the design cycle time.

Summarizing, the aim of this work is “A model of the performance characteristics of a control system that increases the understanding of the relations between hardware and software parameters, such that in early design stages enough confidence is gained to be able to iterate through the design choices with a short cycle-time.”

Modeling approach

In this chapter an approach is presented to make a model according to the aim mentioned before. Although there are techniques, like the ones presented in (Florescu 2006) and (Verhoef 2006), that enable analysis and prediction of the performance of a system before its actual realization, they are not largely used in industry because of their conservativeness or problems to scale with the dimension of the system. Each method makes a trade-off between the time spend to make such a model and the accuracy of the results. Many things influence the performance of a system. In figure 8.1 an overview is provided of typical factors that determine the performance. Four layers are considered:

The lowest level is the *hardware platform* that influences the performance through processor speed, bandwidth and access latency. The efficiency at which it can make use of the memory bandwidth is increased by a memory cache. However, this makes the performance less predictable and more dependent on what exactly runs on the processor.

The next layer is typically the *operating system*, including a scheduler, which takes care of resource sharing by handling task switches and interrupts, and can provide advanced inter-process communication. Then there might be another layer, the *middleware* or services that typically provides *services* and abstractions. The top-layer is the application itself. This application might be modeled entirely with the help of the middleware layer, but usually also contains direct RTOS calls and might directly access the hardware.

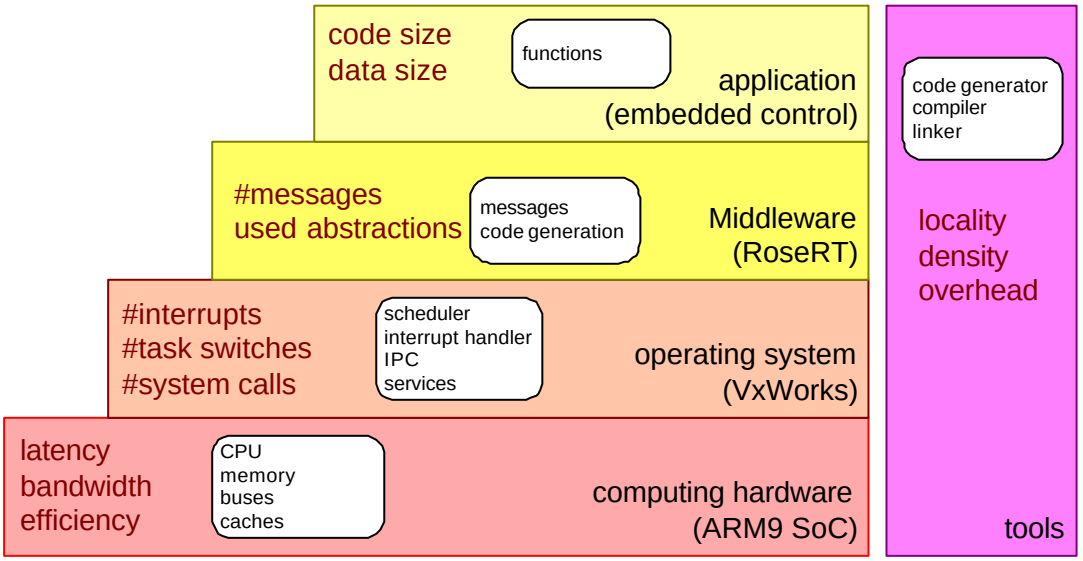


Figure 1. Important layers when considering software performance.

The performance of the entire system depends on how the higher levels use the lower levels. On the vertical bar in Figure 1 the *tools* are mentioned, like compilers and linkers but also code generators of the middleware that can have a huge influence on the performance.

The modeling approach is to consider these layers and to characterize the important aspects

of all these layers with quantifiable parameters. Ideally, the model will be a formula in which the performance (execution time) of the application is expressed as a function of the middleware, RTOS and hardware parameters. The middleware again can be expressed as a function of the RTOS and hardware and the RTOS as a function of the hardware alone. Unfortunately, some characteristics on the lower levels are dependent on the higher levels. For example, the efficiency of a processor is boosted by the use of caches, but the higher levels and tools determine what the influence of the cache will be. Despite it is hard or impossible to estimate these influences accurately, it will be shown that it is possible to create useful insights in the performance.

The case under study

In the next paragraphs, the embedded control software of a printer / copier will be taken as a study object; it will be used for measurements and modeling. The embedded control consists of roughly two parts: a hard real-time part and a soft real-time part. The hard real-time part is the lower level that takes care of things like motor controllers, heater controller, and paper transport; it directly interacts with the environment. The higher layer (soft real-time) is in charge of planning: it receives requests to print or scan one or multiple pages and then makes a detailed planning for these sheets. The planning considers the availability of all functions, like paper path, finisher and printing process.

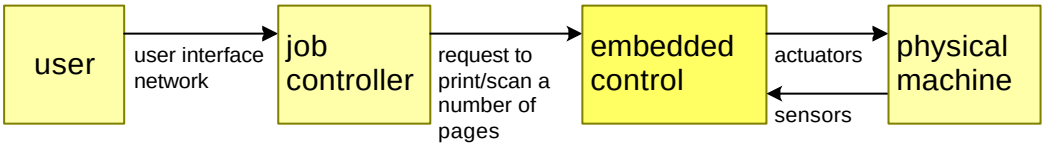


Figure 2. Positioning the embedded control in the printer.

Once this planning or allocation is ready, it is communicated to the lower level control, which will execute it and report back on success or error conditions.

In our case study, the control software runs on a microprocessor (ARM9) on which the VxWorks operating systems is also running. The aforementioned hard real-time tasks are all executed in a periodical task that is called every 2 ms.. This task has a high priority to make sure its behavior is very predictable. The other tasks (like allocation, error handling etc) run as VxWorks threads with lower priority. Most of the control software is generated from RoseRT and uses an extra abstraction layer, the RoseRT runtime system. This runtime system (RTS) includes a mechanism to handle messages between capsules (objects) and handles the execution of state machines that are part of the capsules. The RTS and the application can be spread over multiple threads (each capsule has its own thread) or combined in one.

So, when the system is running, the hard real-time task will interrupt the other tasks every 2 ms and run until completion (of course much less time than 2 ms). The other tasks will only run in the processor time that is left, and typically take longer to finish.

Characterization of the layers

As proposed, the model will be a function that relates the performance of the application to the other layers. For each layer it is possible to measure or calculate a few characteristics. These characteristics can be used to evaluate the performance of the control software as a whole.

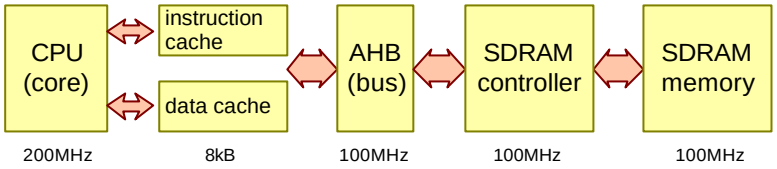


Figure 3. Simplified structure of the SoC showing parts relevant for code execution.

Characterization of the hardware platform. Figure 3 shows the architecture of the chosen system-on-chip (SoC) with ARM9 core. The CPU core runs at a maximum speed of 200 MIPS (Million Instructions Per Second), but because the latency and bandwidth of the memory is much slower this speed will only be reached when all instructions and data are in cache. The system has a two-level memory hierarchy, with a level-1 instruction and data cache and external SDRAM. The cache has 8 words per cache line and 4 sets of 64 cache lines each, resulting in 8kB for instruction and data cache separately. The SDRAM memory and controller have a maximum bandwidth of 100 MHz. Figure 8.4 distinguishes external and internal latencies. Internal latencies are between the CPU core and the SDRAM controller, external latencies are between SDRAM controller and the external memory.

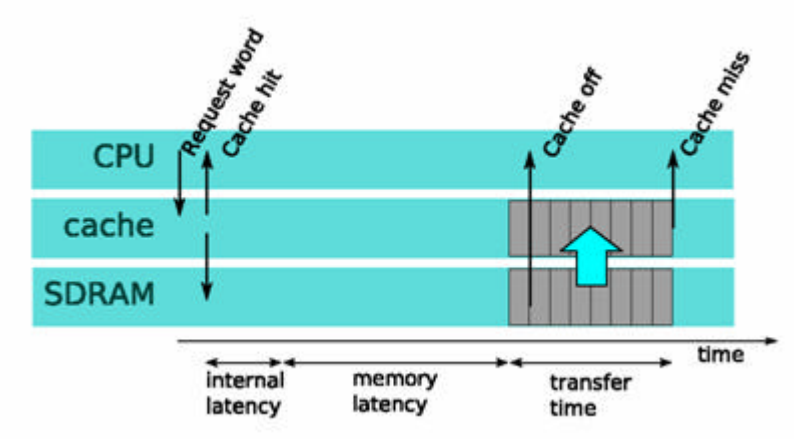


Figure 4. The time for fetching an instruction varies depending on cache setting and availability in cache.

In the case of a cache miss, whole cache lines are fetched at once, which leads to an additional transfer time from memory of 8 memory clock cycles. The CPU-core includes a five stage execution pipeline, the third stage is the execution stage.

Equation 1 is a simple formula for the time it takes to execute a piece of code.

$$T_{exec} = N_i * T_{cpu} * CPI \qquad (1)$$

Where:

N_i Number of instructions in piece of code

T_{cpu} One CPU cycle $1/f_{clk}$

CPI Average cycles per instruction.

The formula can be further refined by specifying the average *CPI* more accurately. When instructions and data are available in the cache, the *CPI* of that instruction will be equal to the one specified in the datasheet of the CPU. Depending on the instruction, it will take 1 to 3 CPU cycles. A branch, for example, typically takes 3 clock ticks because the contents of the pipeline becomes invalid. When the cache does not contain either the instruction or the data (or both), the CPU will be stalled until it is available. Fetching from memory is slower, because the memory

bus is slower, with a factor N_{div}, than the CPU clock. Accessing the memory results in an additional latency; this latency includes amongst others the so-called CAS-latency and is in total N_{lat} memory cycles. Formula 1 can be refined by splitting the instructions, N_i, in instructions that are in cache, N_{fast}, and instructions that are not in cache, N_{slow}.

$$T_{exec} = N_{fast} * T_{cpu} * CPI + N_{slow} * T_{mem} * (N_{lat} + N_{penalty}) \quad (2)$$

Where:

T_{mem} One Memory cycle $T_{mem} = T_{cpu} * N_{div}$

N_{div} Factor between memory and CPU speed

cache setting	measured time [CPU cycles]
normal	815 to 3.9k
flushed	3.9k
off	18k

Table 1: Measured exectution time for 800 NOPs with different cache settings.

The penalty time, $N_{penalty}$, will be explained later on. In order to measure those (combinations of) latencies, 800 individual instructions (eg NOPs) are executed multiple times. This program can be run with different settings of the cache. When the cache is on, eventually all instructions will hit in the cache. This results in a hit rate of 100 %. When the cache is flushed before the execution of the program, all the instructions have to be fetched again (8 at a time, so 100 fetches) from memory. Effectively, this results in a hit rate of 87.5 %. When the cache is disabled, it needs to fetch all instructions (800 times) from the memory separately. This corresponds with a hit rate of 0 %. The resulting execution times for the different situations are measured and listed in Table 1.

Figure 5 shows how the instructions are fetched and executed for different settings of the cache. It is shown that executing instructions is done parallel to transferring them from memory to cache. When all fetches hit in the cache (1 in Figure 5), an instruction is executed every CPU clock, there are no latencies. In the case that the fetch initially misses, the instruction is fetched together with 7 other instructions (2). As soon as the first one is in the cache, it can be executed (3), the latency is 23 CPU cycles. The next sequential instruction can only be executed when it is transferred from memory that is why it is 1 memory clock cycle (2 CPU clocks) later. When the next instruction results in a cache miss, it is still necessary to complete the transfer of all 8 words before fetching of the next words takes place (4), in this case the effective latency adds up to 38 CPU cycles. In the case that the cache is disabled, a word is always fetched from memory before it can be executed (5), the delay is always 23 CPU cycles.

From the measurements and equation 2, it follows that the latency, N_{lat}, is 23 CPU cycles (or 11 memory cycles). $N_{penalty}$ is used to deal with the different effective latency in the case that not everything is in cache. If the hit rate is 1/8, only one instruction is executed while 8 have been fetched, the penalty in that case is $8 * T_{mem} - 1 * T_{cpu}$. However, if the hit rate is 7/8, the penalty is $8 * T_{mem} - 7 * T_{cpu}$, because those 7 CPU cycles were effectively used to execute 7 instructions in parallel with transferring data from memory to cache. In general: $N_{penalty} = 8 * N_{div} - 8 * HR$, with HR the hit rate.

Note that it depends largely on the type of instructions what the average *CPI* is. For example, instructions are only executed efficiently if the code is sequential without branches. A branch instruction flushes the pipeline and has to wait for the cache line to be filled entirely. For now, the effect of the 5-stage pipeline is neglected: an instruction is assumed to be executed when it is available.

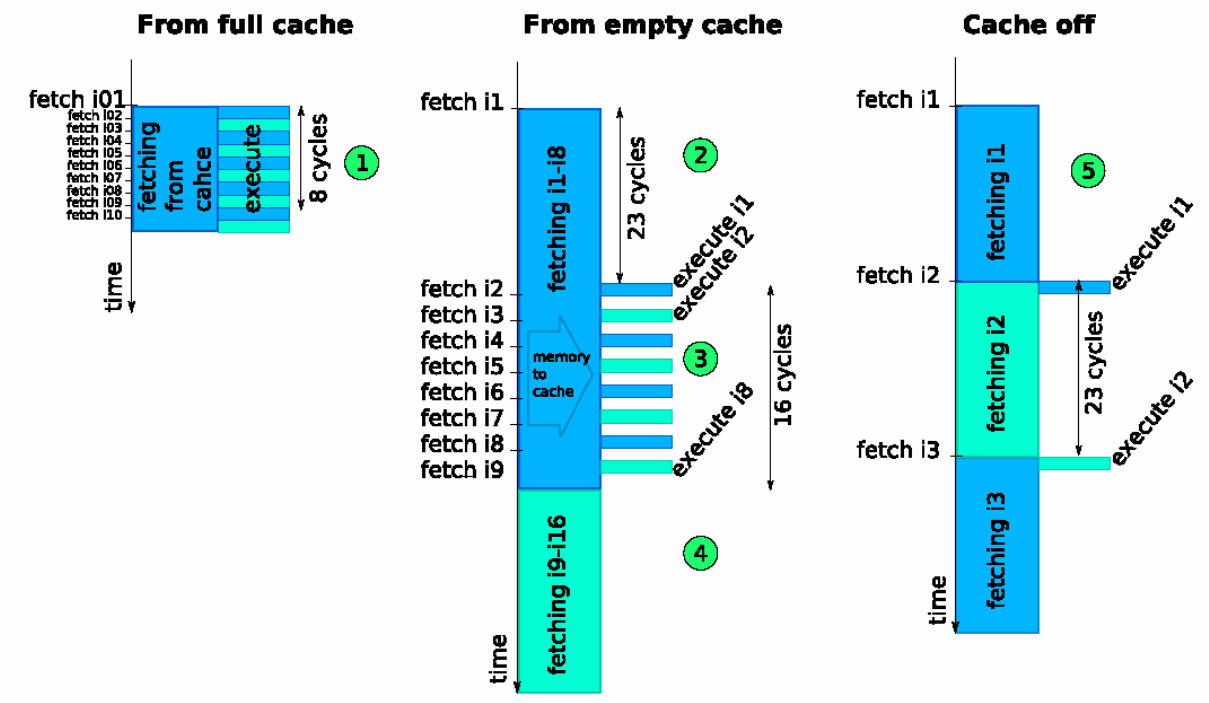


Figure 5. Timing for fetching and executing instructions with caches disabled and enabled.

Measurement method. For all the timing measurements, an on-chip timer has been used. This timer has a resolution of 270 ns. From a few tests of reading the timer register, it has been concluded that the accuracy of the timing method is 200 ns (40x 200MHz-cycles).

Assumptions. In order to simplify the formula, many assumptions were made. These assumptions are important because if they do not hold or cannot be neglected, the formula does not hold and needs adaptation. The most important assumptions are:

Extra latencies caused by the SDRAM are deemed irrelevant. For example switching banks in the memory chips results in higher latencies, but data and code have their own memory banks, and most code is assumed to be very local, reducing jumps over bank boundaries and over SDRAM rows that are 256 words long.

The pipeline of the CPU does not stall, this means no branches (sequential code) and no instructions that have to wait for each others data. When this is not the case, the average CPI will increase, but also the penalty will be different.

Characterization of the RTOS

The RTOS, VxWorks, provides a scheduler that activates and deactivates tasks based on their priority. The scheduler is invoked periodically by a timer and sometimes by tasks through system calls like suspend and semTake. Every time the scheduler is invoked, it has to determine which task to run next and this involves context switching: store the state of the previous task and load the state of the new task. Typically, a profiler like WindView does not show this overhead: it only shows when a task “ends” and apparently the next task immediately starts. With two tasks, like in Figure 7, it is possible to measure the task switching time. Figure 6 shows this graphically: two tasks exist that both run periodically, the timer is read before the suspension of

task1 and after the suspension of task2, which runs at a lower priority. As soon as the main task suspends, task 1 will resume, the cache flush is performed, the timer is read and task 1 is suspended, after which the previously suspended task 2 resumes.

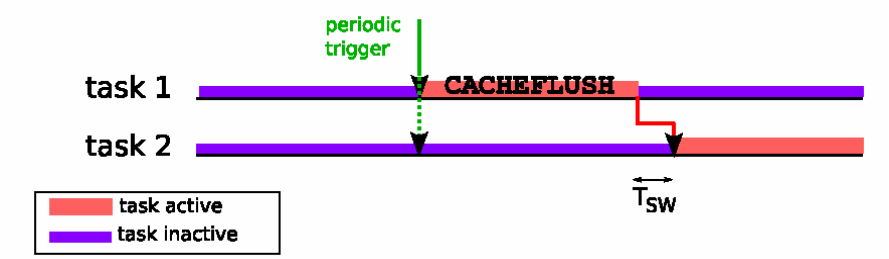


Figure 8.6: To measure the task switching time, we use two tasks that execute sequentially with a cache flush before the switch. The results of the measurements are shown in Table 2. Typically a task-switch will take between 1.6 (best-case) to 20 (worst-case) µs, when caches are enabled and depending on whether the code between the task-switches messes up the cache a lot.

According to Table 2, a task switch with cache disabled takes 10k CPU cycles. 10k divided by 23 cycles per instruction (see Figure 5) is 430 fetches, both instructions and data. Best case 320 CPU cycles are needed, which means that it mostly runs from cache!

```
/* High priority */
void task1( void)
{
  while( 1)
  {
    CACHEFLUSH;
    READ_TIMER( startTime);
    taskSuspend();
  }
}
```

```
/* Low priority */
void task2( void)
{
  while( 1)
  {
    taskSuspend();
    READ_TIMER( endTime);
  }
}
```

Figure 8.7: Example of code used for measuring the task switching time.

cache settings	T_{sw}	
	[CPU cycles]	[µs]
normal	320 to 1.6k	1.6 to 8
flushed	2.8k to 4.0k	14 to 20
off	9.4k to 10k	47 to 51

Table 2: Measured task switch time for different cache settings.

Caching effects by context switching. When a task is interrupted by another task, the current content of the cache is typically worthless: different code will be executed. First the scheduler of the RTOS and then the next scheduled task will be executed by the processor; the cache needs to be "refilled" with relevant contents. Knowing the size of the cache it is possible to estimate the worst-case effect. At most 256 cache lines must be refilled, which gives an overhead of 39 CPU cycles per line: 256*39*5 ns = 50 µs. Therefore, it can be argued that penalty caused by the pre-emption of a task is 50 µs.

Characterization of the middleware: RoseRT

Approximately the same measurement as done for VxWorks with the context switch can be done for RoseRT. Instead of tasks, capsules are considered that send a message (an integer) to each other, see Figure 8. Before sending the message with `messageOut.signal1(0).send()` and after receiving it with `MessageIn`, a timestamp is taken.

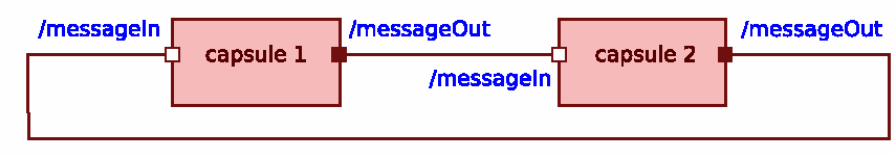


Figure 8.8: Two capsules that send messages to each other.

The scheduling of capsules and messages is done by the RoseRT runtime system, which is linked together with the application code. It can be chosen to make a physical RTOS thread for each capsule, or to map them both on the same physical thread. Depending on this choice, the overhead is different, as shown in Table 3.

Physical threads	*Cache*	Latency
one	normal	[6, 37] µs
	flushed	[33,43] µs
separate	normal	[28, 67] µs
	flushed	[82, 98] µs

Table 3: "Overhead" of sending a message between capsules in different configurations.

Characterization of the application

Formula 2 can be refined more by taking the hit rates of the caches into account, as in formula 3. $N_{penalty}$ has been replaced by its value depending on the hit rate.

$$N_i*((19*T_{mem} - 8 * T_{cpu}*(1 - MR_i))*MR_i + CPI*T_{cpu}* (1 - MR_i)) + N_d *((19*T_{mem} - 8* T_{cpu}*(1-MR_d))*MR_d) \quad (3)$$

Where:

N_i number of instruction fetches
N_d number of data fetches
T_{mem} memory clock cycle time
T_{cpu} CPU clock cycle time
MR_i cache miss rate for instructions
MR_d cache miss rate for data

Therefore, a piece of code (program) can be characterized by values for MR_i, MR_d, N_i, N_d, and CPI. The values for T_{mem} and T_{cpu} are hardware characteristics. For an existing application, the cache miss rate can be measured by executing the code and measure the execution time with caches enabled and again with caches disabled for both data and instruction cache separately. That will result in 3 measurements, obtaining 3 equations for the parameters. Unfortunately, there are 5 independent variables. However, it is possible to determine the set of possible solutions.

The measurements for three cache settings were performed for a part of the soft real-time control code. After analysis, it turns out that there are 3 - 5 more instruction fetches than data

fetches. Furthermore, the miss rate for instructions is between 0 and 5 % and for data between 0 and 18 %. Figure 9 shows the relation for different values of the *CPI*. The values near to 0 % can be confidently neglected, so probably the values will be around 3%miss rate (N_i = 7.7M) for instruction fetches and a data cache miss rate of 10 % (N_d = 1.8M).

Usage of RTOS and middleware. The overhead of the RTOS is mainly due to task switches; during a task switch, the scheduling function is executed. There are at least two task switches every 2 ms because of the hard real-time task. Furthermore there are several other tasks, typically leading to 1500 task switches per second. This number hardly depends on the printing speed. The reason is that after the periodic task always another task is called. One task switch takes worst case 20 µs, the overhead by task switches is therefore at most *1500 * 20 = 30 ms* per second, or 3%.

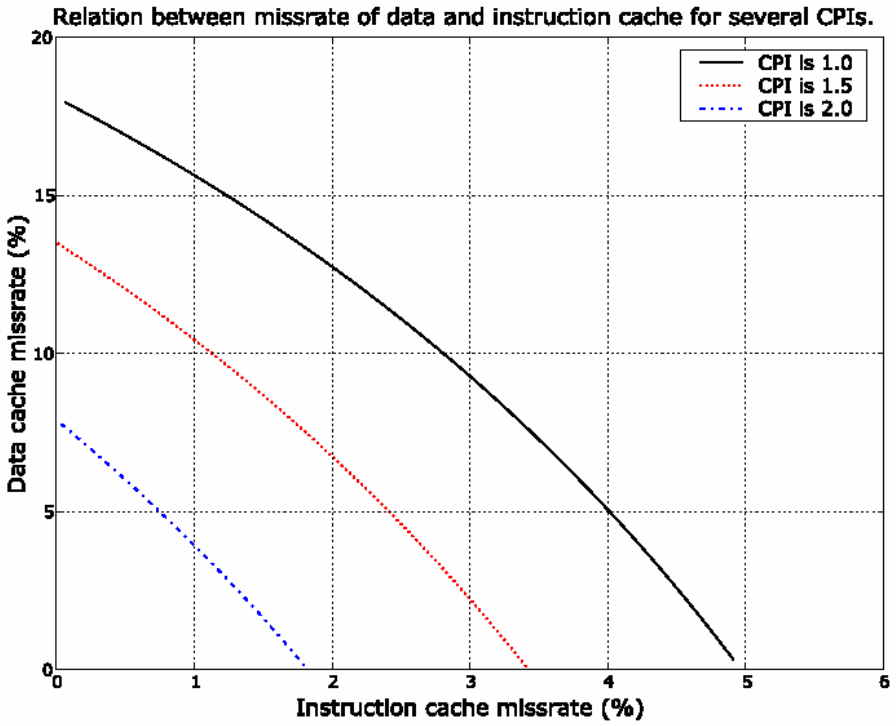


Figure 8.9: Miss rates of data and instruction cache as function of each other, actual value must be on this line, all based on measurements.

The overhead of the middleware is characterized in terms of message overhead. The amount of messages during a print job was measured (typically, this can be done on the target platform if available, but just as well on a simulation on the host). The number of RoseRT-messages per page is 210. Of these messages, 120 are internal in a thread and 90 are between threads, causing extra overhead. With the help of table 8.3, the maximum overhead caused by the messages is calculated to be *120 * 43 + 90 * 98 = 14 ms* per page. Suppose the printer has a speed of 60 pages per minute, then the overhead is at most 1.4 %.

Additional influence of cache. As explained earlier, due to task switches, the cache is spoiled which makes the interrupted task less efficient. In this system, an interrupt occurs every 2 ms, flushing the cache. When a cache is spoiled, it will take at most 50 µs to refill all cache lines and

make the interrupted task run at full speed again. In this case, the harm done by this flushing is therefore at most 500 * 50 = 25 ms per second, thus 2.5 % CPU time. This is the effect of periodic interruption on the soft real-time tasks.

Speed cpu,mem	Estimated time		Measured time
	(CPI=1.0)	(CPI=1.5)	
200,100	107 ms	107 ms	108 ms
100,100	137 ms	161 ms	159 ms
200,50	184 ms	160 ms	178 ms
180,60	162 ms	148 ms	-
160,80	134 ms	134 ms	-

Table 4: Predicted and measured execution time at different clock speed configurations.

Validation

In the previous section, Formula 3 was shown that claims to predict the execution time of an application based on a few measurements on the bare level. With these characterizations, the effect of changing hardware parameters can be estimated. It has been shown already that the effects of task switches and messages can be neglected, although the effect of the parameter changes can also be calculated for them. The effects of four additional hardware platforms are considered (see table 4): the same SoC but with other clock rates for CPU and memory. For two configurations, the measurements are also done for validation. For the configurations of 180 MHz CPU and 60 MHz memory bus, and 160 MHz CPU and 80 MHz memory bus, no validation is done, only a prediction. The latencies of the memories are kept the same number of clock ticks for all configurations. Table 4 shows the measurement results and the corresponding predictions from Equation 3.

It is clear that the correctness of the answer depends highly on the *CPI*. During the previous analysis, a method to correctly estimate the *CPI* has not been considered, but it turns out to be very relevant for the prediction of the execution time.

Conclusions

In the problem formulation we stated that we wanted to come up with a simple model to estimate the performance of the embedded control software. In the following sections some formulas and measurements have been given. As was already said in the problem statement, one of the most important aspects of making a model or a formula, is the insight gained from the formulation. Making a model forces the engineer to be explicit and to quantify and measure relevant aspects, like for example the number of task switches. This is exactly what can be concluded from the case study: insight was gained, but a simple formula that can accurately predict performance on a chosen platforms not yet available. Additionally, the following is concluded:

- A method has been proposed to create a model to estimate the performance of an embedded software application. It is proposed to do simple measurements at each layer. In the particular case, the overhead that can be expected by RTOS and middleware is limited, it is only a few percent. When going to a higher printing speed, only the middleware introduces additional overhead, but it will only become significant at very high printing speeds.
- The method to link application performance to hardware characteristics does provide a lot of

insight in the processor workings. It also gives insight in estimates of characteristics of the application, like cache miss rates and number of instructions. However, the validation shows that especially the CPI is a crucial parameter that has not been addressed thoroughly enough yet.

- In this particular case it has been shown that the overhead introduced by using messages of RoseRT is not very much, approximately 2% of the total. The same argument holds for the time "lost" in context switches. However, in new cases these aspects must definitely be measured and calculated again, it is the only way to be sure.

Furthermore, we like to make the following remarks and recommendations:

- When moving to another platform than the current ARM9, the application itself is not going to change much. However, the execution times will differ. Take for example a Pentium processor. The execution speed of the core is much higher than of the ARM, a factor 10, 2 GHz instead of 200 MHz. The memory bus is typically faster with respect to possible sustained throughput, typically 400 MHz. However, the latency of the memory is not less, it might be even more because of the complexity of a Pentium board, there is a bridge between processor and memory, which will increase the latency. On the other hand, a Pentium has a large L2 (even L3 cache) in which very large parts of the code can reside. The chance that these caches have a miss are very small. Anyway, what needs to be done are the micro-measurements, to get a feeling for the latencies and speeds of the processor board. The effect of the different caches has to be measured and taken into account, this means that it is necessary to estimate the cache misses for all three caching levels.
- There are numerous "details" that influence the execution time of a piece of code. Some of them are parameters of the formulas and can be varied to study the effects. But other things like compiler flags are not in the formulas, but they do influence the execution time. It is important to carefully keep track of all of them, to make them explicit. It would be a good idea to generate a list of relevant parameters to consider. An engineer can then take this list and pick the relevant items for his particular problem.
- Even if information about latencies and bandwidths is available in datasheets or given by another designer, it is worthwhile to do a few measurements. This will give a better "feeling" and forces to validate the implicit model.

References

Ton Kostelijk. "Misleading Architecting Tradeoffs", *IEEE Computer, pp. 20 - 26*, May 2005.

Marcel Verhoef, Peter Gorm Larsen, and Jozef Hooman. "Modeling and validating distributed embedded real-time systems with VDM++". In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Proc. Formal Methods 2006*, volume 4085 of LNCS, pages 147–162. Formal Methods Europe, Springer, 2006.

Oana Florescu, Menno de Hoon, Jeroen Voeten, and Henk Corporaal. "Probabilistic modelling and evaluation of soft real-time embedded systems". In Proceedings of the *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS VI)*, LNCS 4017, pages 206–215, July 2006.

Author Biography

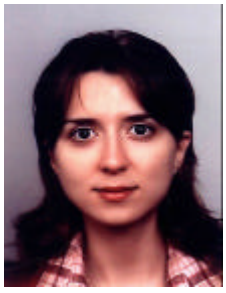
Peter van den Bosch

Peter van den Bosch received his M.Sc. degree in Electrical Engineering from the TU/e (Technische Universiteit Eindhoven, The Netherlands), in 2001. Since 2002, he is a researcher at the research department of Océ Technologies BV. From 2003-2006, he has been working on the Boderc project in collaboration with the Embedded Systems Institute in Eindhoven.

Marcel Verhoef

Marcel Verhoef studied computer science at Delft University of Technology in the Netherlands (MSc, 1993), in the area of computer languages and compilers. He has worked in industry for most of his career. He works for Chess since 1998. He has worked as a systems architect for clients such as the European Space Agency, the Dutch department of Defense, Siemens VDO Automotive, Océ Technologies and Philips. In several of these projects, he applied formal methods. He represents Chess in the BODERC project at the Embedded Systems Institute. He is due to defend his PhD in 2007.

Oana Florescu



Oana Florescu received her M.Sc. degree in Computer Science and Engineering from the "Politehnica" University of Bucharest, Romania, in 2003. Following a six-month internship at Motorola DSP R&D Center Romania, in September 2002, in parallel with her studies, she has started working within their compilers team. A year later, Oana joined the Electrical Engineering Department from the Eindhoven University of Technology for her PhD studies. The focus of her research was on the predictable design of real-time systems within the Boderc project coordinated by the Embedded Systems Institute. During her PhD studies, in the summer of 2006, she went for a three-month internship at IBM Research Laboratory in Zürich, Switzerland. She is due to defend her PhD in 2007.

Gerrit Muller

Gerrit Muller received his Master's degree in Physics from the University of Amsterdam in 1979. He worked from 1980 until 1997 at Philips Medical Systems as system architect. From 1997 to 1999 he was manager System Engineering at ASML. From 1999 - 2002 he worked at Philips Research. Since 2003 he is working as senior research fellow at ESI (Embedded Systems Institute). In June 2004 he received his doctorate. The main focus of his work at ESI is on System Architecture methods and on education of future System Architects. Special areas of interest are: Ways to cope with the exponential growth of size and complexity of systems. Examples of methods to address the growing complexity are product lines and composable architectures. The human aspects of systems architecting (which in itself is a crucial factor in coping with the above mentioned growth). All information (System Architecture articles, course material, curriculum vitae) can be found at: Gaudí systems architecting http://www.gaudisite.nl/