UDC 681.3 https://doi.org/10.15407/pp2020.02-03.149

MODELS OF CONCURRENT PROGRAM RUNNING IN RESOURCE CONSTRAINED ENVIRONMENT

Dmytro Rahozin

The paper considers concurrent program modeling using resource constrained automatons. Several software samples are considered: real time operational systems, video processing including object recognition, neural network inference, common linear systems solving methods for physical processes modeling. The source code annotating and automatic extraction of program resource constraints with the help of profiling software are considered, this enables the modeling for concurrent software behavior with minimal user assistance.

Key words: concurrent programs, program annotation, software modeling, automatons.

У роботі розглянуто моделювання паралельних програм за допомогою автоматів з ресурсними обмеженнями. Розглянуто певні приклади програм: операційні системи реального часу, обробка відео та розпізнавання образів у відеопотоці, вивід у нейромережах, типові методи розв'язку систем лінійних рівнянь для моделювання фізичних процесів. Розглянуто анотування вихідного коду та автоматичне отримання ресурсних обмежень програми за допомогою програм-профайлерів, що дозволяє моделювати поведінку паралельної програми з мінімальною допомогою користувача.

Ключові слова: паралельні програми, анотування програм, моделювання програм, автомати.

В работе рассмотрено моделирование параллельных программ с помощью автоматов с ресурсными ограничениями. Рассмотрен ряд примеров программ: операционные системы реального времени, обработка видео и распознавание образоив в видеопотоке, вывод в нейросетях, типовые методы решения систем линейных уравнений для моделирования физических процессов. Рассмотрено аннотирование исходного кода и автоматическое получение ресурсных ограничений программы с помощью программ-профилировщиков, что позволяет моделировать поведение параллельно программы с минимальной помощью пользователя.

Ключевые слова: параллельные программы, аннотирование программ, моделирование программ, автоматы.

Introduction

The growth of software complexity involves a vast amount of research for improvement of practical software verification, testing theory and testing techniques. Still there is a huge gap between abilities of formal verification techniques and modern needs in fast and less expensive software verification, and this gap is usually bypassed now by increasing efforts for software test cases development. This leads to the "state-of-art" when the software has a big number of errors, but the test case sets and day-by-day use scenarios hide the untested errors and software looks to be "working good enough" and ready to deploy. The motivating examples here are the untouched corner cases, resource allocation problems and erroneous synchronization protocols – all this mix usually goes smooth, but in rare cases fail. Due to rise of complex Linux/FreeRTOS environments anywhere in embedded/IoT/automotive a developer is able to create more complex software using available Linux toolset, so the software complexity increases exponentially. Of course, the software complexity increase leads to the dramatical improvement in product usability but multiplies the number of untested scenarios – but this process is unstoppable due to use demand for improved software usability.

In the paper we consider the cutting-edge modern software applications cases which are affected by software complication growth: software parallelization, concurrent software runs and thread interaction in real-time reactive systems. The concurrent software running on modern parallel platforms is the usual consequence of efforts for minimizing software complexity – as a result a monolithic "serial" program becomes a system of multiple synchronized concurrent threads. The problem here raises that the set of tested thread interaction cases usually covers only well-known interaction scenarios, but in case of resource access denials, lack of resources or temporary device timeouts the software system (sometimes not a system but a mix of threads) fails. In case of parallelized software, the synchronization protocol between execution threads may have drawbacks (mis-synchronization, data loss) and resource access may cause dead-locks. In case of a real-time reactive system, the firmware consists of several concurrent threads under control of simplified real-time operation system without hardware memory management, without blocking thread access into another thread/process memory. In such case even simple non-tested errors (for example while performing pointer arithmetic) cause system fails, so still the stability of software system is guaranteed by hardware watchdog timer.

In the article the theoretical aspects and several practical cases are considered. These cases can improve the management of corner-case and conditions, when the complex software with concurrent threads may fall into erroneous or deadlock conditions due to poor resource (memory, time, etc.) management or simple insufficient resources. The resource constrained automatons are proposed to acts as a model for concurrent processes, and the resource allocation for software models is extracted from results of real software runs. Resource-constrained automaton scalability is addressed as the wrong scaling of software models leads to impossibility to use the models in real life.

Problem statement

Concurrent thread run is not limited only to parallelization of some computation scenario, more beneficial is concurrent run of processes for utilizing available resources. As example – parallel build of source files in project, parallel processing of web site pages or the most popular last years — mining of cryptocurrency.

The well-known Moore's "law" predicted that the computational capability of computers increases twice per eighteen months. This contradicts the serial execution nature of the 95% of developed software, which benefits only 5% speedup with each modern processor generation – it is impossible to speed up serial-fashion program but it is possible to run many independent programs. The "silver bullet" for the computer industry looks to be the parallelization of software, but the economical side of the parallelization looks to be very (or extremely) expensive. Even worse, the math software parallelization methods are studied and evaluated long time ago, and current common computer architectures limits the parallelization gain of software by cache memory structure, size and memory cache to main memory bus bottlenecks. So many computational methods are limited in internal parallelism for 2-4 concurrently working CPUs, but the shortcoming here is that the off-the-shelf CPUs for year 2020 have 8 cores, so the computation capability of middle-level processor exceeds usual software abilities to employ the proposed level of parallelism.

The high parallel computing capability may be easily employed by handling multiple processes, which is common e. g. for modern video content processing – where the other resources (network, memory) utilization is highly predictable. In case of running different types of processes the real parallelism is limited by existing memory and disk I/O – it is a common problem that running more processes may slow down all the system as memory/disk I/O starts to be a bottleneck.

The proper balance of resource capabilities of a system is an important task for CPU clusters as the resource balanced system has additional benefits for user: 1) less processing time, as we do not need extra memory-disk-memory data offloading operations; 2) improved financial efficiency, as hardware resources are employed efficiently and electric and cooling bill is less than in unbalanced case; 3) overall software system can be scaled for "bigger" (in terms of computational power) computing power with less efforts.

In this paper we consider technologies – from formal models to hardware simulations which allows to annotate and characterize resource use of parallel (and serial) applications, targeted to work on clusters or simpler computers with many CPUs. Later this annotation allows to balance concurrent processes execution at some hardware platform. One of target software packages for such kind of parallelization optimization is the weather forecast software from National Meteorologic Institute [1], which is a good point of parallelization and further efficient execution on a parallel computer or cluster. The goal of the forecast package optimization is the continuous use of the software in "software-as-a-service" manner, providing the heavy numerical computations with optimized resource allocation.

Software resource consumption modeling

The first question which raises here is "why we need to account resources if some software can run in virtual environment on some server". The question is correct if we are approaching to parallel software from business side. Virtual environments such as Docker (www.docker.com) allows to run some application in a sandbox creating an illusion that all system resources are manageable by a program. The resource management is passed to supervisor side, and – if several applications start to consume extra resources simultaneously, the system performance will be degraded quickly. (Here different techniques invented to offload highly-loaded computers are not discussed). So the system administrator should reserve resources for the worst system load case. It should be noted that virtual environments are not the proper tool for parallelization, parallelization here works only for simple cases, such as running compilers for big source bases, but even in this case disk I/O is the performance limiting factor. Virtual machines management cases work best for providing unchanged environments (which tends to be obsolete in a half of a year) for specific software with great effort savings on application support, until the software system can be changed completely to the next generation one.

The second question which raises here is "how to employ resource consumption data to day-by-day tasks and how to use it to improve application performance". Taking back to Moore's law we see that the way to improve application performance is to employ more transistors on chip, and the number of transistors on chip grows faster than linear last 30 years (but not exponential due to technology and financial limitations). Therefore, these transistors need to be loaded with some useful workloads otherwise there is no sense in technology improvements. But even this dramatical technology improvements still does not allow to do some modern things with appropriate speed. Neural network inference suffers from low computational speed in all artificial intelligence (AI) tasks. Video processing in real time in HD resolution requires enormous amount of computational capacity. Real-time industrial systems are severely limited in real-time environment sensing. Let us consider several motivating cases.

The good modern case is video processing for streaming cameras – each camera streams an encoded video, which should be got on receiver side (network bandwidth resource is consumed), decoded (hardware video decoder resources are employed, which capacity is limited on a chip), pre-processed (computational capacity grows with image resolution, as example switching from VGA 640x480 resolution to FullHD needs 6-8x computation resources improvement, 4K-format image requires 4 times more resources than FullHD), processed inside computer vision functions (computational requirements raises linearly with picture resolution) and post-processed. Video cards, which are widely used to process images, have fixed amounts of memory; fixed number of executional blocks, special video

processing units (decoders and encoders), limited I/O bandwidth on external memory bus. Each video decoding thread, video filtering blocks, neural network based image recognition blocks requires some defined amount of video memory (depends on actual video stream resolution), some allocation of general purpose computational units and specialized units, memory bus bandwidth allocation. For the most cases resource consumption requirements are changed over the time. So even for proper video card computational load we need to solve – speaking formal way - quite complex dynamic programming task.

Another motivational (and modern) case is the resource scheduling for real-time (semi real-time) embedded applications. Employing embedded Linux (Yocto – yoctoproject.org) for low-power processor had enabled complex Linux-based environments or real time quasi operation systems (e.g. FreeRTOS – freertos.org) for embedded chips (with clock rates up to 1 GHz), which able to process data streams with sampling frequencies exceeding dozens of megahertz. For such cases data streams generation and network traffic cannot be fit into static memory allocation concepts, so applications may temporary allocate extra resources and may get into deadlock on a platform with limited number of resources. The worst scenario here is that the system may hang in dead lock for infinite time instead of hard reset and continuing working.

The reasonable way to avoid resources deadlocks is the prediction of possible resource allocation scenario for a set of concurrent tasks. The main idea of limiting overall resource consumption is that the real-life concurrent tasks are synchronized some way and therefore peak resource consumption intervals have no intersection in time. For non-synchronized tasks the system resource pool should be as large as the sum of maximal resource consumption of each concurrent task. For synchronized tasks their resource consumption should be analyzed in time domain to form a roadmap for resource consumption for each task and predict resource consumption for the whole system. The formal apparatus for such kind of analysis was initially presented in [2] and extended for resource allocation in [3]. It should be noted, that we are not considering only hard real-time systems, where the execution time of operations is strictly determined. Between synchronization points the tasks are not limited to clock-accurate instruction simulation – instruction execution performance may vary and time between resource allocation and deallocation may differ up to 20 % run to run. This fact involves probabilistic analysis of task execution and resource consumption.

The model formalization may be done using timed I/O automatons [4] or the program execution modeling algebra [4]. Exact practical aspects of real-time software modeling are considered in [6], this paper highlights typical software structures which can be efficiently modeled using such algebra. Each *transition* between states is enabled only in case, if all necessary resources are allocated. So, for a distinct resource-constrained model each transition has a corresponding resource requirement, which should be meet at concurrent execution. The program (thread) execution forms a temporal sequence of resource constrains.

Program execution models with resource automatons

The basic question for every software modeling (using temporal logic, timed automatons, imitational simulations via mathematical apparatus or specialized simulations, i.e. network simulations) is the modeling scale. It is impossible for the most cases to simulate a program at basic blocks [7] level, but this is unnecessary. The automaton-based model of program execution should keep program basic properties, which potentially may cause the deadlocks. Let us consider several cases of common program scenarios in the previously mentioned areas:

Case 1. Parallel programs, derived from serial ones by parallelizing loops and utilizing multiple processor cores have the main goal to shorten execution time. The automaton description describes memory resources, used for parallelized loops, dynamic memory allocation and deallocations, synchronization points, mean relative execution times for loop nests, typical execution time for program parts. The good samples of parallelized program kernels are e.g. solving linear algebraic systems (conjugate gradient method) and gas dynamics calculations. The sample automatons for conjugate gradient method case is shown at figure 1.

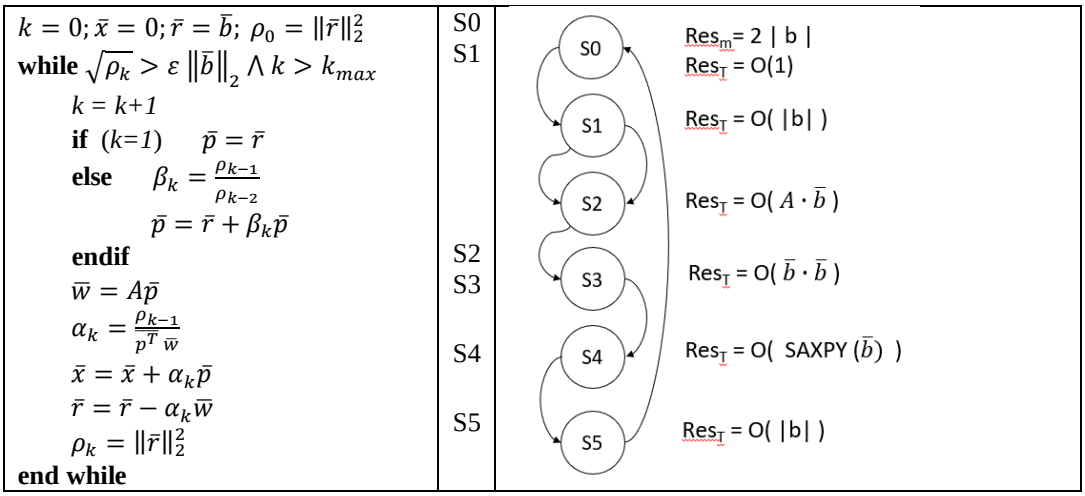

Figure 1. An automaton for conjugate gradient method case

The conjugate gradient method algorithm derived from [8] consists of six main states where computationally intense or memory access intense operations are provided: S0-S5. Computationally intense operation means that the significant part of computations inside the software module (0.1% - 1%) was made during move from this state to the next state. Memory access intense operations means that more than 25% of cache memory storage was updated during the operation. Each state is annotated with resource annotation Res_m and Res_T – for memory and time resources. Memory resource annotation defines how many memory resources are (de)allocated at the state, time resource annotation defines the time which is necessary to process the transition between the automaton states (the time necessary to compute the operations). Time annotation is proportional to the data touched during the operation.

It should be noted that the time annotation is not measured by the number of memory accesses or computations. It is measured by the typical operations computed during the transition. At figure 1 typical operations are: vector copy, scalar multiplication of vectors, matrix-vector multiplication, SAXPY [8]. For these operations the execution time is mainly defined by CPU cache configuration, cache renewal policy and cache misses. The operation run time depends hardly on the fact if the processed data fits into the cache memory and if the computation operation has the necessary data resident in cache just before the computation starts. So the run time depends on dataset size, cache memory configuration, operation type and cache state, but not on the number of operations and memory accesses.

Case 2. Annotations for concurrent programs/threads should indicate memory allocations, synchronization points, mutex sections and I/O accesses. Thread execution times may vary due to different system load (OS interventions, I/O bandwidth and blocking resources by other programs). For specific non-shared hardware cases (for example video cards equipment – coding/decoding units, etc) mutex sections are used, which is annotated by special resource type Res_D. A

The typical example of annotated video processing pipeline automaton is shown at figure 2.

There are two automatons here – the outer one initializes the inner automaton. This two-level structure is required because the outer automaton needs to allocate memory and device resources, as if the resources are insufficient all other operations are useless. The inner automaton shares memory and device resources and requires the device resource to be shared during the defined amount of time. In practice this allows to model various scenarios for device resource allocation and model the scenarios where the video card resource will be used the best way.

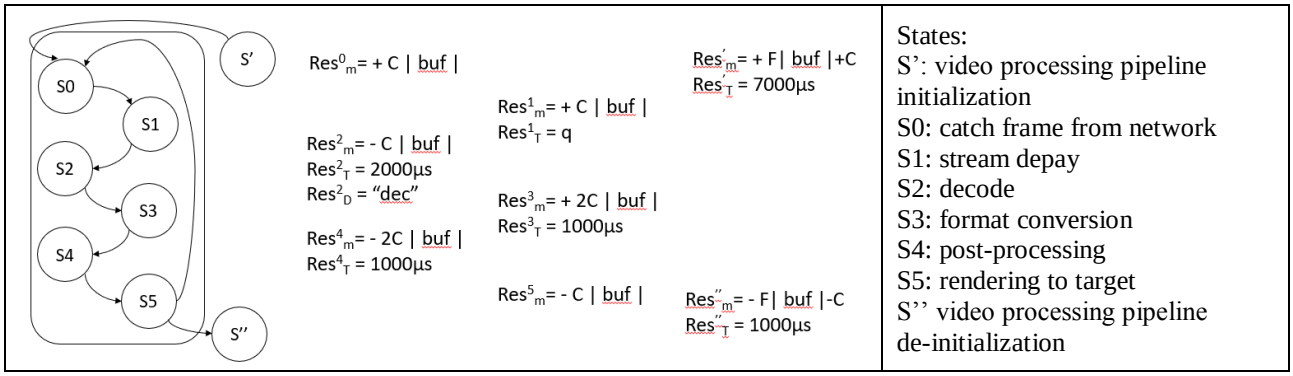

Figure 2. An automaton for video processing pipeline case

This case shows us three types of resources: memory (R^i_m) which define positive and negative quantities, reflecting actual memory budget for automaton parts; time (R^i_T) which defines time necessary to complete a transition to the next state and blocking resource (R^i_D) – usually a hardware resource which needs to be allocated exclusively to complete transition to the next state. It should be noted that timing, memory and blocking resource constraint concepts looks very similar to compiler code generation technology for basic blocks [7], where concurrently compiled operations compete for resource allocation, but usually greedy methods fail to optimize resource allocation.

Case 3. Real-time firmware. The main difference (if compared to (1) and (2) cases) are the big number of uninterruptible execution sections (if compared to other automatons where the actual execution times are automatically extended) and precise execution timings – except parts where software interrupts handlers should do their work. Here one of the real life verification goals is the ability of all system threads to fit into processor execution time and share processor resources without time conflicts. Nowadays the waste of the significant part of processor time in 99% matters is allowed and is not a big problem, but on the other hand the reaction speed for embedded (medical, autonomous driving) applications is critical. Resource and timing conflicts may cause the real-time system to go out of strict timing with severe damage of other system components. Another point is the software upgrade of existing hardware system and the computational capability of system should be investigated. The real-time system analysis is the extremely important, but this case needs to be reviewed in deep and this is out the scope of this paper.

Case 4. Neural network inference is also a corner-stone case for computational capability analysis. The network inference pipeline consists of sequence of computation blocks with precisely defined memory and computational complexity resources, synchronization points are easily constructed using the network structure description. The automaton for classical image segmentation case [9] is shown at figure 3. The main goal of execution pipeline analysis is the maximal use of computational resources and specialized devices.

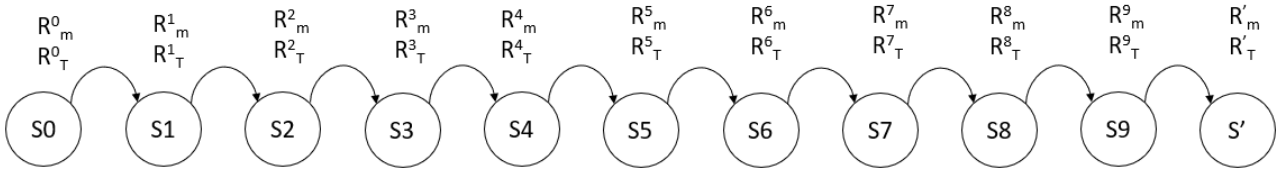

Figure 3. Automaton: segmentation neural network

The figure 3 shows the usual execution sequence for neural network "forward" step. Memory and resource constrains are not trivial, as S5 state depends on data computed during S4 state, S6 on S3, S7 on S2, S8 on S1, and finally S9 on S0. Timings also are unique for each state. For video cards additional dependencies are introduced, which heavily rely on hardware units, memory hierarchy and compiler options for each particular case.

After showcasing various automatons for some practical tasks another question raises – the real annotated automatons need information about resource allocation (memory, processor time, specific device resources). Even if annotation is somehow simplified - there is no need to annotate all processors instructions, but only meaningful (in terms of various resource consumption) blocks of code, but still we need semi-precise quantities for annotations. Let us consider how to get precise information about resource consumption.

Resource consumption modeling

Among the used resources there are resources, which amount can be obtained statistically and resources, which utilization should be known from supervisor. For example, if a program is executed on a videocard, many utilization resource schemes are possible: the program can be executed using different number of devices and sometimes is it not possible to obtain information about real amount of consumed resources. Still there are many resource types which utilization can be calculated using statistical information about program run.

In order to analyse the software we need to model it using a rough model based on automaton model. The rough model needs annotating with resource consumption, which is has enough details to model a software system. One of reliable sources for annotations are software analysis systems, let us consider it at Cachegrind [10] example.

Initially Valgrind&Cachegrind (below in the text it will be referenced as "Cachegrind") were developed as a software for checking memory leaks and modeling cache memory use effects for different processors. Basically Cachegrind simulates program execution instruction-by-instruction, using instrumented heap manager. At the first glance this looks useless, but even modern processors have very limited number of tools for memory debugging – usually they can watch only 4 or 8 small memory regions and their abilities are additionally limited by operating system. As memory leaks debugging is run usually after fixing most of middle and low severity bugs in software, so the execution time is not the issue here. Cachegrind allows to trace operating system calls and memory space by controlling each memory access. At instruction execution level Cachegrind has exhaustive statistics about used computational power. The program, which is run and analysed under Cachegrind control is called instrumented program.

Cachegrind technology also enables precise modeling of processor cache memory, DRAM behavior and interprocessor communication. The most used is cache memory simulation, there practically all possible memory configuration may be evaluated for existing software. As instruction optimization does not benefit as much as memory optimization (except pipelining in deep nested loops) – even for modern SIMD instructions, which just quadruple memory bus load – the precise cache simulation highlights the data traffic between CPU code and cache memories. An example of cache memory behavior is presented at figure 4.

```
      Ir         Dr         Dw I1mr     D1mr      D1mw I2mr     D2mr      D2mw
       .          .          .    .        .         .    .        .         .  double array_sum(double a[N][N])
       .          .          .    .        .         .    .        .         .  {
       .          .          .    .        .         .    .        .         .    int i,j;
       .          .          .    .        .         .    .        .         .    double s;
       1          .          .    .        .         .    .        .         .    s=0;
   3,001          .          .    .        .         .    .        .         .    for(i=0;i<N;i++)
3,004,000         .          .    .        .         .    .        .         .      for(j=0;j<N;j++)
1,000,000 1,000,000          0    0  125,000         0    0  125,000         .        s += a[i][j];
       .          .          .    .        .         .    .        .         .    return s;
       1          1          0    0        1         0    0        1         .  }
       .          .          .    .        .         .    .        .         .
       .          .          .    .        .         .    .        .         .  int main(int argc, char** argv)
       1          0          0    1        0         0    1        .         .  {
       .          .          .    .        .         .    .        .         .    double a[N][N];
       .          .          .    .        .         .    .        .         .    int i,j;
   3,002          0          0    1        0         0    1        .         .    for(i=0;i<N;i++)
       .          .          .    .        .         .    .        .         .    {
3,004,000         .          .    .        .         .    .        .         .      for(j=0;j<N;j++)
       .          .          .    .        .         .    .        .         .      {
1,000,000         0  1,000,000    0        0  125,000    0        0  125,000         a[i][j]=0.01;
       .          .          .    .        .         .    .        .         .      }
       .          .          .    .        .         .    .        .         .    }
```

Figure 4. Cache memory simulation example

The figure shows the number of cache misses into instruction cache and 1st and 2nd levels of data cache. The 2nd data cache miss rate is the actual traffic to memory (need to multiply it by cache memory line size). At figure 4 it is clear that function *array_sum*() produces 125000 1st and 2nd level cache read misses (memory read requests) and 125000 1st and 2nd level write misses (memory write requests). This leads to 8MB traffic to cache memory, which is the real dataset size and practically annotates the code block. The memory access pattern in fact defines the time which the function needs to run and the function run time may be calculated here precisely as the memory access time dominates among other factors which define the function run time. It should be noted that the number of memory accesses depends as polynomial on the cardinal number (here – N) of the algorithm, so can be interpolated and the correct value of memory traffic can be calculated for practically all cases of program execution.

Other good sample for accounting memory resources is neural network inference procedures. The most network operations are matrix multiplications [11] with well-known memory footprint and communication patterns (for parallel matrix multiply versions). For shared memory multiprocessors matrix multiplication is parallelized using e.g. BLAS (netlib.org/blas) also with well-known footprint and interprocessor traffic. Therefore memory timings, memory traffic, memory footprint are well-known and as the memory multiplication is mostly memory traffic bounded - the execution time is highly predictable. Synchronizations between inference computations for each level are well-defined, so the resulting automaton with annotations described the process execution with high precision. Neural network inference, represented as an execution graph usually is acyclic which highly simplifies automaton analysis.

The only annoying issue for Cachegrind is the highly increases instrumented program execution time. If Cachegrind simulation time is compared to a simulation based on hardware processor event counters (used by all modern profilers such as Vtune), the latter gives practically no overhead, but gives less accurate information for process time line, as the event counter sampling is not made instruction accurate (as statistical sampling methods are used). Cachegrind may introduce up to 100x overhead. On the other hand, for applications, which are run under the same dataset size (e. g. such as forecast package [1]) the resource constraints may be calculated only once for all future execution scenarios as resource utilization is not changed, or re-calculated statistically, so only one or several profiler runs are enough to get necessary annotation data.

Using resource annotations for resolving resource conflicts

Let us consider a simple sample of two concurrent threads which periodically request for some amount of shared resource, and the resource pool capacity is less than the sum of all requests from concurrent threads. The left part of figure 5 shows the two automatons for two concurrent threads with annotations for shared resource R_X demand. For figure 5 the condition is: $R_X^1 + R_X^1 > \|R_X\|$, so the simultaneous entering states S1 and S1' causes deadlock at memory allocation. In order to avoid the deadlock the states S1 and S1' should not be processed simultaneously – entering states S1 and S1' should be synchronized, processing S1 and S1' states should be mutually exclusive. The right part of figure 5 shows the source automaton after transformation, where additional states T1, T1', T2, T2' are introduced. These states act as semaphores to synchronize entering S1 and S1' states. Such kind of synchronization can be built on the top of old good standard system libraries with semaphores/mutexes, etc (pthreads, for example). The more fruitful point is that the tools and instrumentation, which allows to introduce synchronization points may process automatically using annotations into the source program text and transform the source code without need to work at compiler (code generation) level. Even more – these annotations can be reused in case of changed data set size, using Cachegrind-gathered information.

The resource allocation problem is not unique for resource-constrained automatons and was studied well in compiler theory, where main hardware processor execution paths are considered as resource-constrained automatons and the code generator should choose among multiple instruction execution combinations only one which utilizes maximum system resources [7]. As the representation of resource set is very similar to resource representation for code generation for some processor – for resource-constrained automaton, the set of compiler-related techniques for resource allocation may be considered, for example Linear Scan for Registers, which uses heuristics for decreasing time of efforts necessary for analysis of resource constrained automation work.

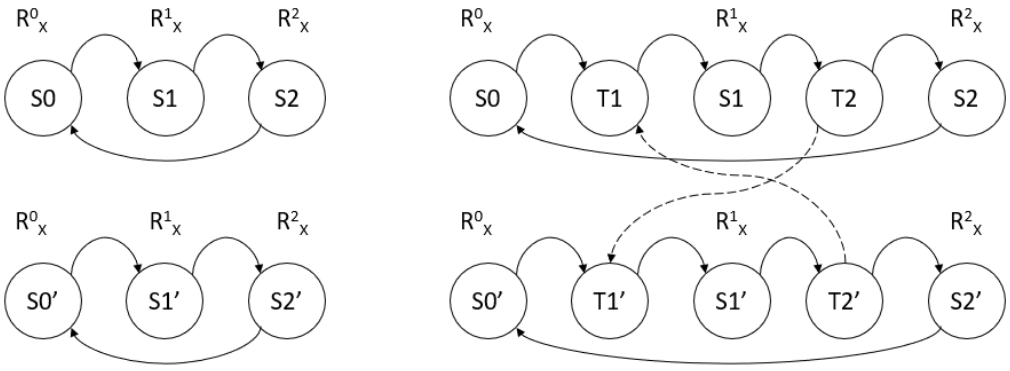

Figure 5. Automaton transformation

The constraint resolving may used quite simple greedy methods, this works well for automatons from figure 5 (left part). Potentially or really conflicting states are enclosed with a pair of states (T1 and T2, T1' and T2'), which works in pair and provides a semaphore object to orchestrate resource use. The method is extensible, as such type of a semaphore can be extended for any number of parallel threads. It should be noted, that the automatons should be verified for absence of resource leakage, so that processing over a closed path should not increase the amount of consumed resources.

Still resource constraints handling for some states in a set looks to be ineffective for bigger automatons, as state insertions make the overall software model to complex. The solution is the use of virtual machine, which analyses the resource consumption between the state transitions and continues work only for some automatons in the set. Anyway the virtual machine is impossible without automaton analysis and analysis of automaton closed paths, as greedy approach for processing automaton schemes may be cause deadlocks here. The virtual machine concept analysis will be considered in future papers.

Source code annotations enables the used and added states and/or the virtual machine. Annotations include automaton states description and resource consumption information. Similar to modern automatic documentation systems (i.e. well-known Doxygen) annotation are placed in comments with some specific header, for example the possible automaton description – the automaton similar to left ones from figure 5 – follows (the left column is a description in the source code, the right column included the description explanation):

/// State S0: R[Mem]=+2MB; R[T]=+4MC /// State S1: R[Mem]=+124MB; R[T]=+120MC /// State S2: R[Mem]=-126MB; R[T]=+30MC /// Trans: S0>S1; S1>S2; S2>S0	State S0: memory consumption 2MB, time – 4 million cycles State S1: memory consumption 124MB, time – 120 million cycles State S2: all memory marked as free, time – 30 million cycles Transition table

This description can be automatically extracted from code comments by a utility, similar to C/C++ preprocessor or a command-driven stream editor (similar to Linux *sed*). In the source code the beginning of each state is marked with a simple comment line similar to the following:

/// State S0 code start	Start of code (place synchronization primitives here)
/// State S0 info start	Start of code where Cachegrind info should be gathered
/// State S0 info end	End of code where Cachegrind info is gathered

This "code start" annotation marks the place where the synchronization primitives can be placed (inserted) by automatic preprocessor or a virtual machine should reserve resources for the state processing. The synchronization primitives and virtual-machine based thread control can be based on the top of some old good standard POSIX library, e.g. *pthreads*. In case if the automaton granularity was chosen correctly the time overhead for synchronization is neglible.

The precise information about resource consumption can be derived from Cachegrind information. Specific annotations points to code places where Cachegrind gathers performance information should be summarized to get memory traffic consumption metric (L2 misses). The sample Cachegrind summary (shown at figure 4) and quite a simple utility can be used to extract e.g. L2 misses information (the same *sed* stream editor utility from Unix may be used) and calculate summary information for an automaton state. The extracted information can be placed into external definitions file (C-style header file) to use while considering automaton resource consumption descriptions (in virtual machine) _– and organized as a record of resource consumption values.

The major benefit of the presented automaton description is that the automaton analysis can be done in less or more automatic way, with minor human interventions. The timing and memory consumption information can be gathered semi-automatically via profiling tools – Cachegrind is only one from the wide class of profiling software which can be used to evaluate resource consumption. The virtual machine tracks automaton model run and only in case of incorrect or complex model the user intervention is necessary. The simplest automaton analysis can be done in greedy ways, more complex techniques such as forward resource scans may be tested. If the automaton will be expressed in some standard way, for example [6], the existing verification software can used for automaton evaluation at least for time metric, which is useful for verification and consideration for real-time software. The practical algorithm is not considered now in the paper – it's the topic for the following research for resource constrained automaton analysis for real software automatons.

Conclusion

Various practical aspects of optimizing resource consumption for heavy loaded computer systems are considered in the paper. The formalism defined in [3] can be used to describe the concurrent running threads for example as automatons with resource consumption annotations. Several practical examples of expressing software processes/threads as automatons are presented. A methodic for gathering resource consumption information using state-of-the art profiling tools was presented. Also, a methodic for annotating the program code with automaton descriptions

was provided which enables to control the automaton execution via a virtual machine. Automaton verification can be provided using existing verification tools and model environments.

The next point of research is the efficient use of verification tools based on existing frameworks. It is possible to reinvent the wheel and to create another verification tool, but the more useful activity is to re-use existing tools and frameworks and this is the goal of the upcoming research.

References

1. Prusow V.A., Doroshenko A.Yu. Modeling natural and technogenic processes in atmosphere. Kyiv: Naukova dumka, 2006. 541 p.
2. Ahlem Triki, Jacquez Combaz. (2013) Model-Based implementation of Parallel Real-Time Systems. Verimag Research Report TR-2013-11
3. Rahozin D. A resource limited parallel program model. Problems in Programming. 2008. N 4. P. 3–10.
4. Kaynar D., Lynch N., Segala R., Vaandrager F. The Theory of Timed I/O Automata. Morgan&Claypool. 2011. 137 p.
5. Peter Hui and Satish Chikkagoudar. (2012) A formal model for Real-time Parallel computations. In Proc. Of FTSCS-2012. P.39-53.
6. Basu A., Bogza M., Sifakis J. Modeling heterogeneous real-time components in BIP. In SEFM. 2006. P. 3–12. IEEE Computer Society.
7. Steven S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann 1997.
8. Golub G., Van Loan C. Matrix computations. Johns Hopkins University Press, 1996.
9. Minaee S., Boykov Y., Porikli F., Plaza A., Kehtarnavaz N., Terzopoulos D. Image Segmentation Using Deep Learning: A Survey. arXiv:2001.05566v2
10. Weidendorfer J. (2008) Sequential Performance Analysis with Callgrind and KCachegrind. In: Resch M., Keller R., Himmler V., Krammer B., Schulz A. (eds) Tools for High Performance Computing. Springer, Berlin, Heidelberg. P. 93–113.
11. Chetlur S., Woolley C., Vandermersch P., Cohen J., Tran J. cuDNN: Efficient Primitives for Deep Learning. Nvidia, 2014. arXiv:1410.0759v3

Література

1. Прусов В.А., А.Ю. Дорошенко. Моделювання природних і техногенних процесів в атмосфері. Київ: Наукова думка, 2006. 541 с.
2. Ahlem Triki, Jacquez Combaz. (2013) Model-Based implementation of Parallel Real-Time Systems. Verimag Research Report TR-2013-11
3. Rahozin D. A resource limited parallel program model. Problems in Programming. 2008. N 4. P. 3–10.
4. Kaynar D., Lynch N., Segala R., Vaandrager R. The Theory of Timed I/O Automata. Morgan&Claypool. 2011. 137.
5. Peter Hui and Satish Chikkagoudar. A formal model for Real-time Parallel computations. In Proc. of FTSCS-2012. 2012. P. 39–53.
6. Basu A., Bogza M., Sifakis J. Modeling heterogeneous real-time components in BIP. In SEFM. 2006. P. 3-12. IEEE Computer Society.
7. Steven S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann 1997, ISBN 1-55860-320-4
8. Голуб Дж., Ван Лоун Ч. Матричные вычисления. Пер. с англ. / Под ред. Воеводина В.В. –М.: Мир. 1999. 548 с.
9. Minaee S., Boykov Y., Porikli F., Plaza F., Kehtarnavaz F., Terzopoulos D. Image Segmentation Using Deep Learning: A Survey. arXiv:2001.05566v2
10. Weidendorfer J. Sequential Performance Analysis with Callgrind and KCachegrind. In: Resch M., Keller R., Himmler V., Krammer B., Schulz A. (eds) Tools for High Performance Computing. Springer, Berlin, Heidelberg. 2008. P. 93–113.
11. Chetlur S., Woolley C., Vandermersch P., Cohen J., Tran J. cuDNN: Efficient Primitives for Deep Learning. Nvidia, 2014. arXiv:1410.0759v3

Received 20.02.2020

About the author:

Dmytro Rahozin,
PhD,
Senior Scientific Researcher,
Publications in Ukrainian journals – more than 10.
Publications in foreign journals – 2.
http://orcid.org/0000-0002-8445-9921.

Authors' place of work:

Institute of Software Systems National Academy of Sciences of Ukraine,
03187, Kyiv-187,
Academician Glushkow Avenue, 40.
Phone: (38)(044) 526-21-48.
E-mail: Dmytro.Rahozin@gmail.com