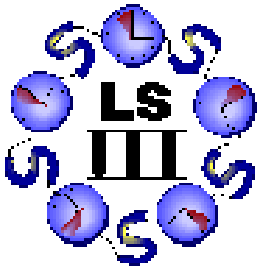
Final Report

SAADI
Integrated Design Approach of Adapted Schedulers

Project Group 424



Project Group
at the Computer Science Department
of the University of Dortmund

March 2003

Instructors:

Prof. Dr. Horst F. Wedde,
MSc Muddassar Farooq,
Dipl.-Inform. Mario Lischka

Students:

Ammar Mohamed Ammar,
Abdulwhab Bador,
Christian Bockermann,
Rolf Hipler,
Ioannis Karagiorgos,
Piotr Kasprzak,
Tobias Malbrecht,
Simon Muras,
Mattias Stöneberg,
Markus Wübben,
Erdal Yigit

This document was created with LaTeX 2_ε.

Contents

List of Figures

List of Tables

Listings

1 An Introduction to SAADI

by Markus Wuebben

Operating systems can now be found in many electrical devices ranging from desktop PCs, server systems to PDAs and mobile phones. All these devices are embedded in different environments and need to satisfy different requirements. While server systems may want maximum task-throughput, desktop PCs should provide maximum interactivity and mobile devices set special focus on energy-efficient computing. Schedulers are one of the core elements of operating systems affecting its behaviour more than any other component. Good task-scheduling can support if not enforce the demanded behaviour of the operating systems. Unfortunately even schedulers in modern operating systems still provide only a limited set of scheduling capabilities. This motivated us to start a two semester project at the University of Dortmund called *SAADI - An Integrated Design Approach to Adaptive Schedulers*. In the scope of this project we lift schedulers to a whole new level by making them *adaptive*. We interpret adaptability in the sense that schedulers are sensitive to the requirements a task-set of a system has.

This report depicts the results of our work.

In chapter 2 we present the theoretical foundations of our approach, focus on the *Scheduler Description Language* in chapter 3 and in chapter 4 we present our testing methods. Chapter 5 outlines the implementation details. Finally we conclude our work in chapter 7.

1.1 Tasks Challenges

by Christian Bockermann

There are many different challenges to be met in the development of an efficient scheduling algorithm. As mentioned above computers run in different environments which are mainly defined by the software they run. SAADI approaches the task to find well-optimized scheduler by classifying tasks (i.e. threads) of software upon their requirements. Taking a closer look at an internet browser one can specify different tasks of that browser: the User-Interface-thread, a thread for downloading data and perhaps an interactive multimedia thread playing a video. These threads define different criteria they need to meet while running. The UI-thread for example waits for user-input while the download-thread runs in background. The download- thread generates a lot I/O, while the multimedia-thread is CPU-bound computing a lot of frames. The later one needs a certain amount of CPU time in a periodical manner to support a certain frame-rate.

Since most of these facts are known when the software is being written the first approach of SAADI is to define a language by which they can be expressed. This is defined as the SAADI Description Language, a XML-based language described in chapter 3. As the requirements of software are defined the next step is to find an algorithm which is optimal to schedule all these tasks. Since one can not define all the tasks which are to be run at a certain time it is hard to find an optimal scheduler. SAADI uses a hierarchical scheduler tree to combine benefits of several different schedulers into one scheduler

tree. By testing several randomized scheduler trees which are then evaluated and improved by a genetic algorithm (see chapter 2), SAADI tries to find a scheduler tree which meets all requirements in an efficient manner. This leads to three basic parts of the SAADI framework: The genetic algorithm with the SDL-parser, a Linux kernel framework to easily implement various scheduling hierarchies and a set of scripts to start the Linux Trace Toolkit used to measure the performance of the generated hiearchy. The basic steps to explore a set of randomly generated hierarchies (i.e. a *population* of hierarchies) are simple:

1. Specify software requirements using the SDL-language.
2. Generate a basic population of hierarchies.
3. Create a test environment for a hierarchy of the population, compile a kernel and measure its performance.

The last two steps are to be repeated until a hierarchy meets the requirements and achieves a good fitness. The results of the tests are stored in a relational database. This makes it easy to find an appopriate hierarchy for a similar specification later without the need to go through the whole testing procedure again.

2 Theoretical Approach

2.1 An Introduction of SAADI

by Muddassar Farooq

The emergence of internet market towards the end of the last century has not only revolutionized the PC market but also helped in establishing new paradigms for business processes. Hence we experienced enormous advancements in processor technology, its accompanying peripherals on the one hand and in different *user centered* software applications exploiting these advancements on the other hand. This resulted in a huge paradigm shift about the usability of personal computers (PC) from their traditional use in automating task to a cardinal component of a computer supported collaborative work (CSCW) environment. Nowadays in a CSCW environment, a user could start a download from a remote site, attending a multimedia conference at the same time, listening to light music on his CD, and taking notes of minutes of the meetings, reading/sending important emails in the mean time and running a complex simulation in the background. A flux of these evolving applications is making the processor usage pattern of the applications unpredictable because of greater diversity in the needs of the users and their CSCW environments. These conflicting but coexisting scheduling requirements make it an uphill and difficult task to have a generic scheduler that could easily adapt to the needs of such a wide spectrum of the user community.

Traditionally the field of scheduler design has been enthralled with operating systems designers whose major concern was to allocate a fair share of processor time to each task so that their response time is bounded but no guarantees are given to turn around time or deadline failure rates. On the contrary, designer of real time schedulers try to meet the deadline of the task with rate-monotonic and earlier deadline first (EDF) algorithms, but offer no support for conventional applications. Yet third community of designers dedicated considerable efforts to hybrid approach in which the scheduling behavior is realized using hierarchical scheduling paradigm. This hierarchical scheduling paradigm enabled schedulers to schedule a mix of heterogenous applications. However in all these approaches, *users's community* was never involved in the design cycle of schedulers (see Figure 2.1). This leads to not only unsatisfactory experience of the customers in the *information technology* business on the one hand but also added to the frustration of dedicated application developers who were unable to provide scheduling policies for even for those applications that are developed by themselves. This non-existence of communication channel between a user of PC and the scheduler of PC provided us the grist for the mill in doing research in an interesting field of *user centered scheduler design (UCSD)*.

This paper describes our SAADI (Integrated Approach for Adaptable Schedulers) project. In this project we try to overcome this anomaly by proposing a *user centered scheduler design* approach so that a user is actively involved in the design cycle of a scheduler (see Figure 2.2) and hence occupy a pivotal position in formulating scheduling policies. In our framework SAADI, we try to involve a user by asking him to specify syntactic structure of his applications of interest, their relative importance and value to him, and semantic specification for scheduling requirements for the operating system

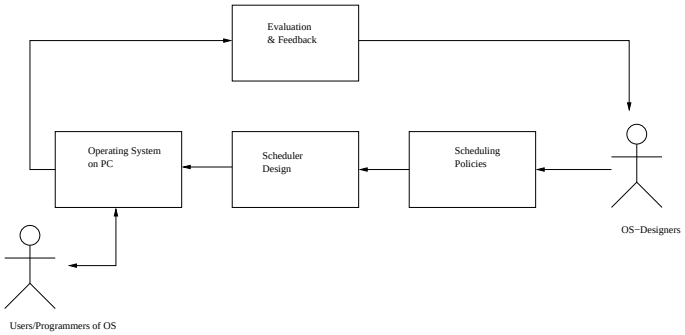


Figure 2.1: Traditional Scheduler Design Cycle

scheduler. By user of the scheduler in this paper we mean that a person who interacts with the OS to achieve his goals. Hence in our framework we take care of three types of user: A novice user, An advance user and Developer. A novice user is a person who has virtually little or no programming experience but has a knowledge about different type of applications that he will be running on his machine. An advance user is the one who has some experience in programming and has the ability to describe his applications and their scheduling requirements. A developer is an advance user who is involved in developing applications to be run on a machine and hence has the ability to describe the demands of his applications at each thread level.

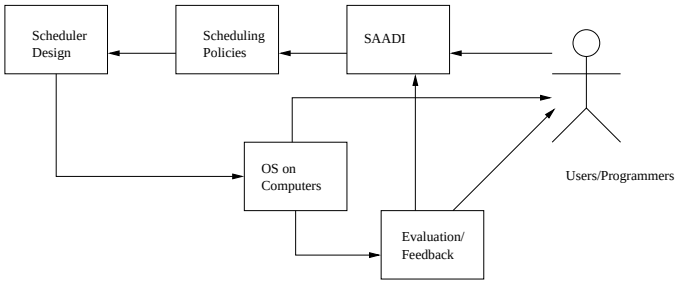


Figure 2.2: SAADI Scheduler Design Cycle

2.1.1 Formal Representation

Abstract Scheduler Model is at the moment formulated as Meta Hierarchical Scheduler (MHS) a variant of HLS.

Definition 1 (Schedulable Object) *A Schedulable Object $S_j^{S_r}$ can be an application (A_k), a thread k of an application j (T_{kj}) or another scheduler, that will be scheduled by its parent scheduler S_r.*

Definition 2 (Guarantee) *A guarantee $g_{S_k}^{S_r}$ is a contract between a scheduler S_r and a schedulable object S_k regarding the allocation of CPU time by S_r to S_k according to the definition of* guarantee

as long as $g_{S_k}^{S_r}$ is enforced.

Definition 3 (Guarantee Set) *The following guarantee set G is incorporated in our ASM $G = \{PS, PSBE, RESBS, RESPS, FP, ALL, NULL\}$ where PS means proportional share, PSBE proportional share with bounded error, RESBS means a periodic reservation, RESPS a probabilistic reservation and ALL means that complete CPU time, NULL means no guarantees.*

Definition 4 (Scheduler Set) *Our ASM supports at the moment a scheduler set $S_{r_{(g_i)}}^{(g_o)}$ where $S_r = \{PS^*, RESBS^*, FP, O(1)\}$, $g_i \in G$ is an input guarantee that $s \in S_r$ receives from its predecessor scheduler $s_p \in S_r$ and g_o is an output guarantee that scheduler $s \in S_r$ provides to its successor scheduler $s_c \in S_r$ .Where RESBS,TS,PS FP schedulers provide their respective guarantees $g \in G$, and O(1) is a standard Linux scheduler by Ingo Molnar.*

Definition 5 (Schedulable Set) *A schedulable set S_l is defined as*

$$S_l = \bigcup S_r \quad \cup \bigcup_{k=1\ldots n} A_k \quad \cup \bigcup_{\substack{i=1\ldots m \\ j=1\ldots l}} T_{ij}$$

Where A_k is an application with single thread of execution and T_{ij} represents thread j of multithreaded application i.

Definition 6 (Schedulable Relation) *A schedulable relationship SR_{lm} between schedulable object m and scheduler l exists if*

1. $\exists \quad l \prec m \rightarrow$ *m is a successor of l*
2. *m receives on this arc guarantee g_l^m provided by scheduler l*

Definition 7 (Root Scheduler) *A root scheduler $S_{root} \in S_r, S_{root} \notin S'_r$ is the one that gets as input guarantee $S_{root_{(g_i)}} = All$ i.e it has all of CPU time at its disposal for allocating it to its successor schedulables. S'_r refers to a scheduler set whose members can not become a root scheduler. At the moment $S'_r = \{O(1), FP\}$.*

Definition 8 (Meta Hierarchical Scheduler) *A structure $MHS = \{S_{root}, G, S_l, SR_{lm}\}$ in our ASM is called a* Meta Hierarchical Scheduler, *where*

(1) S_{root} is the root scheduler of the current hierarchy (Definition 7)

(2) G is the guarantee set supported in the current hierarchy (Definition 3)

(3) S_l is the schedulable set (Definition 5) synthesized from SAADI-SDL in a manner that satisfies following conditions

$$\bigcup_{k=1\ldots n} A_k \prec m_k = \emptyset$$

$$\bigcup \quad T_{ij} \prec m_{ij} = \emptyset$$

$$j = 1 \ldots l$$

This implies that applications with single threads or threads of multithreaded applications could only appear at the leaf of a hierarchy.

(4) $SR_{lm} \subseteq S_r \times S_r$ *subject to following conditions*

a) $l \in S_r, m \in S_r, \forall l, m : l \neq m$

b) $\forall l, m, \exists SR_{lm} \Rightarrow \nexists SR_{ml} \wedge \nexists SR_{km} :: k \in S_r, \forall k \neq l$

c) $if \quad m = RESBS \Rightarrow \exists SR_{lm} \Leftrightarrow l = RESBS$

d) $if \quad m = PS \Rightarrow \exists SR_{lm} \Leftrightarrow l \in \{PS, RESBS\}$

e) $\forall l, m, \exists SR_{lm} \Leftrightarrow S_{m_{(g_l)}} = S_l^{(g_l)} = g_m^l$

Definition 9 (Guarantee Operator) *A Guarantee Operator f_g is a mapping between requirements of applications specified in SDL R_{sdl} to a Guarantee Set (Definition 3) supported in SAADI*

$$f_g : R_{sdl} \mapsto Pot(G)$$

where Pot(G) is a set of all subsets of a Guarantee Set supported in SAADI.

Definition 10 (Guarantee Word) *A Guarantee Word W_g of length n is a set of tuples of guarantees with length n. Let us assume that a SDL has n applications*

$$S = Perm(f(A_1) \times f(A_2) \times ... \times f(A_n)), f(A_i) \in Pot(G), 1 \leq i \leq n$$

where Pot(G) is a set of all subsets of a Guarantee Set supported in SAADI.
Now these tuples could be easily mapped to a set of terminal characters (T)

$$W_g : S \mapsto T^*, W_g(a_1, ..., a_n) \mapsto \{a_1 \# a_2 \# ... \# a_n\}$$

where a_1,a_2,a_3....a_n are terminal character for corresponding guarantee type and # are separation characters.

Definition 11 (Meta Scheduler Language) *A Meta Scheduler Language MSL in our ASM has the form MSL = (T, V, S, P) where*

- *T, the finite alphabet, from which the language is formed (terminal characters) i.e a terminal character for a guarantee type*
- *V, a finite alphabet, with T disjunct set of help characters (variables)*
- *$S \in V$, the starting symbol*
- *P, a finite set of derivation rules (productions)*

  With the help of MSL we can build a MHS that meets that satisfies the requirements R_{sdl} of a set of applications.

Definition 12 (Meta Scheduler Grammar) *A Meta Scheduler Grammar (MSG) provides a formal description of tuples of MSL (Definition 11) with the help of which we could build a MHS (Definition 8) in our ASM that satisfies the requirements of applications specified in SDL.*

The form of our grammar MSG = (T, V, S, P) can be found at **??**.

2.2 Regehr

by Piotr

It is clear that our approach to the problem of scheduling of tasks implies a modular, non-monolithic framework that is able to handle very diverse application and user demands. For example one user wants a scheduler that optimally supports his use of a PC as a personal workstation involing mostly running interactive and multimedia applications. Therefore the scheduler needs to support time-sharing and soft-realtime scheduling policies. Another user might want to use the PC as a web-server requiring high throughput (and no soft-realtime scheduling capabilities that might have a negative impact on the throughput) and even load isolation (e.g. if the user is a provider who wants to use the web-server for various customers and also wants to guarantee a fair use of CPU-time between them - a sensible wish). A general monolithic scheduling algorithm can not fulfill all these demands in an optimal way. Instead it is just optimized for a general usage scenario that was anticipated by the developers (if the user is lucky the developers of the scheduler also tried to minimize worst case behavior in some not-so-common situations). Another obvious shortcoming of this traditional approach to scheduler design in multi-purpose operating systems (called MPOS from now on) is also the inability to optimize or adapt the scheduler to new application requirements that might emerge in the future and are unforseen at the point of development (which is quite common in the fast moving computer industry). So after looking through various research on scheduler models that are able to support extremly diverse scheduling behavior we finally found a perfect fit: hierarchical scheduling. The theoretical foundation of our project is mainly based on the research done by John Regehr about the composition of schedulers into hierarchies.

2.2.1 Hierarchical Scheduling

The main idea of the hierarchical scheduling approach is to arrange simple, well understood schedulers (e.g. fixed-priority, reservation based, time-sharing) into a hierarchy to derive the complex scheduling behavior needed to support a given set of applications. A simple example that illustrates this idea and also allows a first glimpse at the power of this model is shown in figure 2.3.

Schedulers are shown as boxes with their type denoted inside. The arrows symbolize the ”scheduled-by” relation. The root scheduler of this simple hierarchy is a fixed-priority scheduler (FP) that schedules a cpu-reservation based scheduler (RES) with high and a ”joint scheduler” (J) with low priority. A joint scheduler is a construct that allows us to reuse slack time that was reserved for a scheduler or a task but not used up. It is the only type of scheduler that can be ”scheduled by” more than one scheduler (from a graph-theoretical point of view this construct generalizes the underlying structure of the hierarchy into a directed acyclic graph). The reservation-based scheduler (RES) can provide the soft-realtime requirements needed by the video-player application because it takes precedence in the distribution of CPU-time in relation to the joint scheduler (enforced through the fixed-priority root scheduler (FP)). On the other hand the use of the joint scheduler (J) allows us to guarantee the allocation of a certain amount of CPU-time to the proportional-share scheduler (therefore preventing a pos-

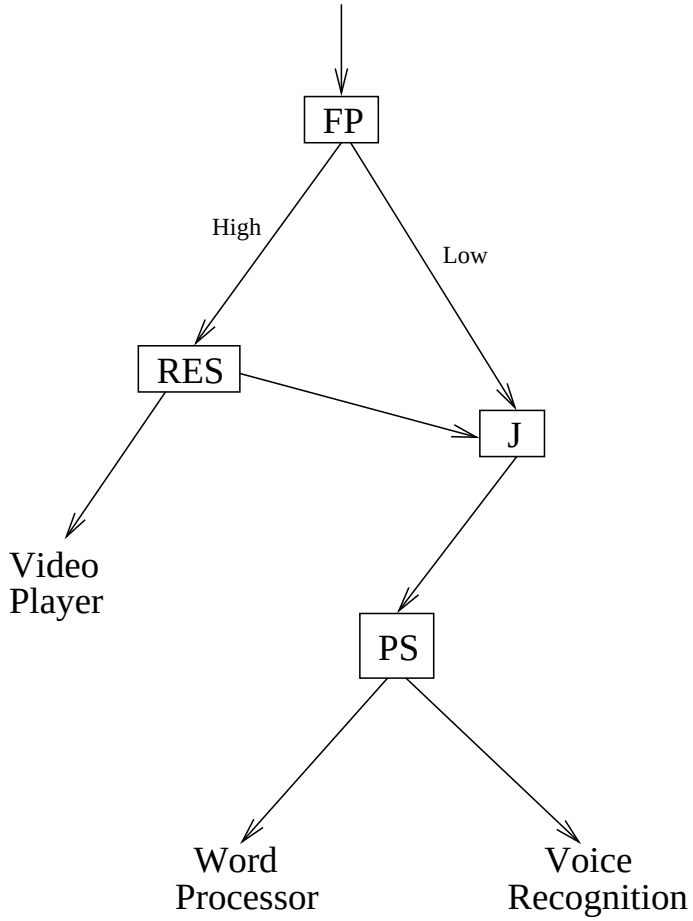


Figure 2.3: Example hierarchy of schedulers

sible starvation of the word-processor or voice-recognition application that could have been induced by the soft-realtime requirements of the video-player application in a traditional MPOS scheduler) and also to use possible slack time in the schedule when the video-player application does not use up its reserved CPU-time amount. So by using a simple reservation based scheduler, a fixed-priority scheduler and a joint scheduler as components and arranging them in a ”sensible” way into a hierarchical structure we were able to solve one major scheduling problem that is often exhibited by MPOS schedulers with some minimal soft-realtime support: starvation of tasks.

2.2.2 Guarantees

We just articulated that we needed to arrange the basic scheduler components in a ”sensible” way to derive a scheduling behavior that satifies our requirements. But what does ”sensible” mean in this context? Clearly the arrangement of the schedulers can not be arbitrarily. Let’s revisit our example hierarchy of figure 1. If for example we scheduled the joint scheduler with high priority and the reservation based scheduler with low priority the reservation based scheduler would not be able to provide the desired soft-realtime scheduling to its child task (it would only be scheduled if the proportional-share scheduler had no tasks to schedule). But how is it possible to analyze and formally show that a

given scheduling hierarchy exhibits the desired behavior? At this point the concept of a ”guarantee” comes into play. A ”guarantee” is a fundamental abstraction of our scheduler model. It is a formal statement describing the allocation and distribution of CPU-time promised to a scheduled object in our hierarchy (task or scheduler) by its parent. Guarantees are independent of a particular scheduling algorithm in the sense that different scheduling algorithms are able to provide the same guarantee. Now it becomes possible to express and analyze scheduling behavior with a set of abstracts: every scheduler in the hierarchy needs an acceptable ”incoming” guarantee from its parent scheduler to provide ”outgoing” guarantees to its children. This relation is determined by the implemented scheduling algorithm. A guarantee conversion system is described in section (2.2.3). From the point of view of the guarantee system the problem of creating a ”sensible” hierarchy that is able to meet the desired requirements can be described as follows: the purpose of the scheduling hierarchy is to convert the ”ALL” guarantee that represents 100% of CPU-time and is provided to the root scheduler of the hierarchy by the operating system into a set of guarantees matching the requirements of the application set. So let’s have a look at the syntax and semantics of the guarantees implemented in our SAADI-framework at this point. The following syntax is used:

GUARANTEE-TYPE [params]

where **[params]** is a comma separated list of numerical values describing various parameters. Integers normally represent time units in the millisecond scale (unless otherwise stated).

The ”ALL” guarantee

”ALL” represents a formal statement about the assignment of 100% of the CPU-time. It is given to the root scheduler of the hierarchy by the operating system and can obviously be used by any scheduler as an acceptable ”incoming” guarantee.

The ”NULL” guarantee

When no guarantee about the amount or distribution of CPU-time can be made (indicating simply a best-effort scheduling) the ”NULL” guarantee is used.

CPU reservation guarantees (”RES”)

CPU reservation guarantees describe soft-realtime scheduling behavior and are especially useful in conjunction with applications that are not able to degrade gracefully when they receive a smaller amount of CPU-time or an inadequate distribution of CPU-time than they require. Most multimedia applications fall into this category (e.g. your favriote mp3-player or CD-burning tool). Our scheduling framework is not able to guarantee hard-realtime behavior due to the missing hard-realtime support in the standard linux kernel. Stolen time due to IO-operations, swapping, interrupts or other operating system activity can not be directly guarded against without a significant change of the appropriate code-paths inside the kernel which is beyond the scope of our work. Nevertheless the 2.5.x (soon to be 2.6.x) linux kernel - the platform for our scheduling work - has been greatly improved in the area of system induced latencies and can handle most real-time applications pretty well. In the following detailed description of the different CPU reservation guarantees ”x” denotes an amount of CPU-time while ”y” denotes a period. Both values are given in milliseconds. Please note also that - following John Regehr’s nomenclature - ”soft” means at least the described amount of CPU-time guaranteed

while ”hard” means excatly the described amount of CPU-time guaranteed. This is not to be confused with the normal meaning of ”hard” and ”soft” in conjunction with realtime scheduling policies. The following types of guarantees are defined:

- RESBS x, y
  Denotes a basic-soft CPU reservation guarantee with amount x and period y
- RESBH x, y
  Denotes a basic-hard CPU reservation guarantee with amount x and period y
- RESCS x, y
  Denotes a continuous-soft CPU reservation guarantee with amount x and period y
- RESCH x, y
  Denotes a continuous-soft CPU reservation guarantee with amount x and period y

As can be easily seen all four guarantee types are constructed by picking either ”basic” or ”continuous” and ”soft” or ”hard”. The meaning of each will be described now.

”basic” CPU reservations The formal semantic of a ”basic” guarantee is that there exists a time t so that for every integer i the receiving schedulable object is promised a CPU-time allocation of x time units in every interval of $[t + i * y; t + (i + 1) * y]$ time units. No guarantees regarding the time t or the the distribution of the CPU-time amount ”x” inside the period-length interval of ”y” are made.

”continuous” CPU reservations ”Continuous” CPU reservations guarantee that for any time t a CPU-time amount of x will be provided in the interval of $[t; t + y]$

This implies that the distribution of CPU-time in each interval of the form $[t+i*y; t+(i+1)*y]$ (as defined in 2.2.2) must be the same therefore leading to a much stronger guarantee: every ”continuous” guarantee is also a ”basic” one the opposite being not true.

”hard” CPU reservations A ”hard” CPU reservations provides the receiving schedulable object with exactly the x time units every period of y time units - and no extra time, even if it would be available in the schedule. This mechanism is meant to constrain the CPU-time usage of applications that can not generate additional value for the user from extra CPU-time.

”soft” CPU reservations ”Soft” CPU reservations express the same guarantee about the provided CPU-time like the ”hard” variant. In contrast to the latter additional CPU-time may be provided to the application on a best-effort basis if available. It is assumed that applications with a ”soft” reservation guarantee can use the extra time to provide additional value for the user.

Proportional-share guarantees

”Proportional-share” guarantees as used in our scheduling framework can roughly be devided into two groups: proportional-share guarantees with bounded error (PSBE) and the weaker proportional-share guarantees without any deterministic bounds (PS).

proportional-share guarantees with bounded error (PSBE) This guarantee type has the following syntax: $PSBEs, \delta$ where "s" denotes the share of the CPU-time promised to the receiving object ($s \in [0..1]$). Please note that the share "s" is absolute rather than relative allowing for an easier, localized analyzing of the resulting hierarchy. The δ-parameter is a bound on the error-term in the provision of the CPU-share meaning that for any time t the schedulable object is guaranteed to receive a $s * t - \delta$ share of the CPU-time (δ is of course dependent on the implemented scheduling policy and may be difficult to calculate).

weak proportional-share guarantees (PS) If no deterministic bound on the just described error-term can be made by the scheduling algorithm a proportinal share guarantee of the following form can be used: PS s
The parameter "s" once again describes a share of the CPU-time ($s \in [0..1]$). But because no bound on the error-term can be provided this share is viewed as an asymptotic value resulting in a much weaker guarantee compared to a PSBE-type guarantee.

2.2.3 Guarantee Conversions

Forms of guarantee conversions

As mentioned above from the guarantee system point of view the problem of finding a suitable scheduling hierarchy can be described as a guarantee conversion problem. We take the guarantee "ALL" and try to convert it into a set of guarantees that is acceptable to our application set. Two distinct forms of guarantee conversions exist in our framework:

- indirect guarantee conversions by a scheduler
- direct guarantee conversions

Guarantee conversions through schedulers

Every scheduler in our framework is charaterized by one or more tupels of the type acceptable guarantee, set of provided guarantees meaning that the given scheduler can locally convert a guarantee of the acceptable type into a set of guarantees denoted as "set of provided guarantees". Note that more than one such tuple can exist depending on the implemented scheduling algorithm. Table 1 provides an insight into the guarantee conversion abilities of different known scheduling algorithms. The syntax used is:

- $A \mapsto B^+$ means that scheduler can convert a guarantee of type "A" into a set of guarantees of type "B". "A" shall be the weakest acceptable guarantee (what "weak" means in this context will become clear after the next section).
- "any" denotes a variable that can represent any guarantee type.

To make the scheduler-based guarantee conversion process more comprehensible for the reader we will discuss some of the presented guarantee conversions in more detail.

Scheduling algorithm	**Guarantee conversion**
fixed priority	$any \mapsto (any, NULL^+)$
limit	$RESBS \mapsto RESBH$
proportinal share	$PS \mapsto PS^+$ $PSBE \mapsto PSBE^+$ $RESU \mapsto PSBE^+$
cpu reservation	$ALL \mapsto RESBH^+$ $RESU \mapsto RESBH^+$
time sharing	$NULL \mapsto NULL^+$
BSS-I, PShED	$ALL \mapsto RESU^+$ $RESU \mapsto RESU^+$
CBS	$ALL \mapsto RESBH^+$ $RESU \mapsto RESBH^+$
EEVDF	$ALL \mapsto PSBE^+$ $RESU \mapsto PSBE^+$
Linux, Win 2000	$NULL \mapsto NULL^+$
Lottery, Stride	$PS \mapsto PS^+$ $RESU \mapsto PS^+$
Resource Kernel	$ALL \mapsto (RESBS^+, RESBH^+)$ $RESU \mapsto (RESBS^+, RESBH^+)$
Rialto, Rialto/NT	$ALL \mapsto RESCS^+$ $RESU \mapsto RESCS^+$
SFQ	$PS \mapsto PS^+$ $PSBE \mapsto PSBE^+$ $RESU \mapsto PSBE^+$
SFS	$PS \mapsto PS^+$ $RESU \mapsto RESBH^+$
Spring	$ALL \mapsto RESBH^+$ $RESU \mapsto RESBH^+$
TBS	$ALL \mapsto RESBS^+$ $RESU \mapsto RESBS^+$

Table 2.1: Guarantee conversions induced through various scheduling algorithms

Fixed-priority Assuming that schedulable-objects use distinct priorities (this can be enforced by admission control) and are scheduled by a preemptive fixed-priority scheduling algorithm, we easily see that the guarantee conversion properties are independent of the provided ”incoming” guarantee. Instead the guarantee provided to the scheduler is just passed on to the child with the highest priority. The other children of the scheduler get only the ”NULL” guarantee (the child with the highest priority starves all other children if it is CPU-bound). The complexity of analyzing the case where the child with the highest priority is not CPU-bound does not warrant any results about possible guarantees that could be made to the other children.

Limit A ”Limit” scheduler is used to convert a soft CPU reservation into a hard CPU reservation. This can be accomplished by keeping track of the amount of CPU-time used by its (single) child per time-period and releasing the CPU if additional, not guaranteed CPU-time becomes available.

CPU reservations The CPU reservation scheduler implemented in SAADI needs to receive an ”ALL” guarantee to provide ”RESBH” guarantees to its children.

Time-sharing Our basic round-robin based time-sharing scheduler can use any guarantee to generate ”NULL” guarantees for its children (implying simple best-effort scheduling behavior). From a formal point of view the weakest acceptable guarantee to our time-sharing algorithm is the ”NULL” guarantee which can be easily converted to from any other guarantee (section 2.2.3).

Direct guarantee conversions

The general theory established by John Regehr shows that it is possible to convert certain guarantee types into other. One trivial example illustrating this possibility is the following: assume we have a ”RES x, y” guarantee given with x denoting the amount of CPU-time and y denoting the period (the distribution of CPU-time inside the period beeing irrelevant). Then it’s easy to see that this guarantee can be ”seen as” (or converted to) a ”PS $\frac{x}{y}$” guarantee making it acceptable to any schedulable object needing to receive a ”PS s” guarantee where $s \leq \frac{x}{y}$. On the other hand a conversion in the opposite direction is not possible because a ”PS” guarantee does not imply any time-based bounds on the distribution of the CPU-time. Table 2 shows the possible direct conversions between the different guarantee types and has the following syntax:

- $\bigcirc$ denotes that no conversion is possible
- $\surd$ denotes that a conversion is possible
- a number enclosed in paranthesis references the theorem from which the presented result has been derived (a missing number indicates that the result is trivial, e.g. ”ALL” $\mapsto$ ”RES”, etc.)

The referenced theorems justifying the different conversions are the following:
CPU reservations of type ”basic” can be converted into ”continuous”, ”soft” CPU reservations:

Theorem 1 *The guarantee RESBS x, y and RESBH x, y can each be converted into the guarantee RESCS x,* $(2y - x + c)$ *for any* $c \geq 0$.

On the other hand ”basic” CPU-reservations can not be converted into ”continuous”, ”hard” CPU-reservations unless x = y (which implies the guarantee ”ALL”):

ALL	√	√	◯	√	◯	√	√	√	√
RESU	◯	√	◯	◯	◯	◯	◯	√	√
RESBH	◯	◯	√	√	◯(2)	√(1)	√(4)	√(5)	√
RESBS	◯	◯	◯	√	◯(2)	√(1)	√(4)	√(5)	√
RESCH	◯	◯	√	√	√	√	√(3)	√(5)	√
RESCS	◯	◯	◯	√	◯	√	√(3)	√(5)	√
PSBE	◯	◯	◯	√(6)	◯	√(6)	√	√	√
PS	◯	◯	◯	◯	◯	◯	◯	√	√
NULL	◯	◯	◯	◯	◯	◯	◯	◯	√
↦	ALL	RESU	RESBH	RESBS	RESCH	RESCS	PSBE	PS	NULL

Table 2.2: Direct guarantee conversions

Theorem 2 *Neither RESBS x, y nor RESBH x, y can be converted into a continuous, hard CPU reservation unless x = y.*

”continuous” CPU-reservations may be regarded as proportional-share guarantees with bounded error:

Theorem 3 *The guarantees RESCH x, y or RESCS x, y may be converted to the guarantee PSBE* $\frac{x}{y}, \frac{x}{y}(y-x)$.

”basic” CPU-reservations may also be regarded as proportinal-share guarantees with bounded error:

Theorem 4 *The guarantess RESBH x, y or RESBS x, y may be converted to the guarantee PSBE* $\frac{x}{y}, 2\frac{x}{y}(y-x)$.

All types of CPU-reservations can be converted into a proportoinal-share guarantee:

Theorem 5 *Any CPU reservation, whether ”hard” or ”soft”, ”basic” or ”continuous”, with amount x and period y may be converted into the guarantee PS* $\frac{x}{y}$.

Proportinal-share guarantees with bounded error can be seen as ”soft” CPU-reservations:

Theorem 6 *The guarantee PSBS s,* δ *can, for any* $y \geq \frac{\delta}{s}$*, be converted into the guarantee RESCS* $(ys-\delta)$*, y or RESBS* $(ys-\delta)$*, y.*

2.3 MHS Generation

by Simon Muras, Mattias Stöneberg

To solve the problem of generating a hierarchy for a given set of application requirements the following procedure is proposed based on context free grammars:

1. Initially requirements of applications are mapped to a guarantee type that could meet the requirements. The concatenation of all these guarantee types formulates a word. Later on we try

to formulate a context free grammar that could generate this guarantee word. Therefor exactly one guarantee type has to be assigned to each application thread which fits the requirements as close as possible(we assume that all guarantees in this word a ordered by the thread‘s id).

2. Using the guarantee conversion rules (direct and based on scheduler types), guarantee types could be converted to another guarantee types and this helps in construction of a context free grammar.

3. The CFG-parse algorithm by Earley can verify if the generated word w, $w \in L(G)$ (= word problem). During this verification, the generated syntax tree, with the help of our grammar rules, is transformed into a tree-based scheduling hierarchy. The Leafs of this scheduling hierarchy represent required guarantee types.

2.3.1 Mapping of the application requirements to a word

The mapping of a set of requirements to a certain guarantee type is obviously deeply dependent on the precision with which the requirement were stated and does not have to be unambiguous. Although at this point of time a random algorithm is used for word mapping, the word ambiguity could be used as an interfacing point to our knowledgebase.

Taking a slightly more formal point of view:

$$f : A \mapsto Pot(G),$$

with A = {application requirements in the SDL} and G = {supported guarantee types in SAADI}. Let n denote the number of the application threads described in the SDL. With

$$S = Perm(f(A_1) \times f(A_2) \times ... \times f(A_n)), f(A_i) \in Pot(G), 1 \leq i \leq n$$

we get a set of tuples of guarantees with length n which can be mapped to a word. The mapping is trivial due to the fact that each guarantee type is represented by its own terminal string which we will see in the next section. Let T denote the set of terminal characters:

$$g : S \mapsto T^*, g(a_1, ..., a_n) \mapsto \{a_1 \# a_2 \# ... \# a_n\}$$

This leads to a valid guarantee word, we will now take a look at the definition of a possible context free grammar.

2.3.2 The context free grammar

A grammar is characterized by:

- T, the finite alphabet, from which the language is formed (terminal characters)
- V, a finite, with T disjunct set of help characters (variables)
- $S \in V$, the starting symbol
- P, a finite set of derivation rules (productions)

Context free grammars are grammars where all productions have the form $A \mapsto v$ with $A \in V$ and $v = \varepsilon$ (empty word) or $v \in (V \cup T)^*$.
Our grammar $G = (T, V, S, P)$ has the form: $T = T_1 \cup T_2$, where
T_1 = { implemented guarantee types (e.g. null, resbs, ps, ...) }
T_2 = { separation character (e.g. # or white space) }
$V = V_1 \cup V_2 \cup V_3$, with
V_1 = { implemented guarantee types (but with a distinct representation compared to T, e.g. NULL, RESBS, PS) }
V_2 = { supported scheduler types, where the acceptable "incoming"-guarantees are also taken into consideration (e.g. for a PS-scheduler: SCHED-PS-PS, SCHED-PS-PSBE)}
V_3 = { ALL }
S = ALL
$P = P_1 \cup P_2 \cup P_3 \cup P_4 \cup P_5$, where
P_1 = { $G \mapsto g \mid G \in$ { implemented guarantee types } $\subseteq V, g \in$ { implemented guarantee types } $\subseteq T$, where G and g refer to the same guarantee type }
P_2 = { scheduler based guarantee conversions (e.g. for a PS-scheduler: SCHED-PS-PS $\mapsto$ PS, SCHED-PS-PSBE $\mapsto$ PSBE) }
P_3 = { rules to connect scheduler types with the different acceptable "incoming"-guaranties (e.g. PS $\mapsto$ SCHED-PS-PS or PSBE $\mapsto$ SCHED-PS-PSBE }
P_4 = { supporting rules to model scheduler based guarantee multiplications (e.g. PS $\mapsto$ PS # PS}
P_5 = { direct guarantee conversions (e.g. ALL $\mapsto$ PS, RESBH $\mapsto$ RESBS) }
The following grammar file is the currently used:

Definition 13 $G = (S, T, V, P)$, $V = V_1 \cup V_2 \cup V_3$, $P = P_1 \cup P_2 \cup P_3 \cup P_4$

T= { *resbs, ps, null, fp* }

V_1= {*RESBS, PS, PSBE, NULL, FP* }

$$V_2 = \left\{ \begin{array}{ll} \textit{SCHED-PS-PS}, & \textit{SCHED-PS-PSBE}, \\ \textit{SCHED-PS-ALL}, & \textit{SCHED-PSBE-PS}, \\ \textit{SCHED-PSBE-PSBE}, & \textit{SCHED-PSBE-ALL}, \\ \textit{SCHED-RESBS-RESBS}, & \textit{SCHED-RESBS-PS}, \\ \textit{SCHED-RESBS-PSBE}, & \textit{SCHED-RESBS-ALL}, \\ \textit{SCHED-TS-PSBE}, & \textit{SCHED-TS-RESBS}, \\ \textit{SCHED-TS-NULL}, & \textit{SCHED-TS-ALL}, \\ \textit{SCHED-TS-FP}, & \textit{SCHED-TS-PS}, \\ \textit{SCHED-LINUX-PSBE}, & \textit{SCHED-LINUX-RESBS}, \\ \textit{SCHED-LINUX-ALL}, & \textit{SCHED-LINUX-FP}, \\ \textit{SCHED-LINUX-PS}, & \textit{SCHED-FP-PSBE}, \\ \textit{SCHED-FP-RESBS}, & \textit{SCHED-FP-ALL}, \\ \textit{SCHED-FP-PS}, & \textit{SCHED-FP-FP} \end{array} \right\}$$

$V_3 = \{$ *ALL* $\}$

$S = \{$ *ALL* $\}$

$P_1 = \{$ *RESBS* $\rightarrow$ *resbs, PS* $\rightarrow$ *ps, NULL* $\rightarrow$ *null, PSBE* $\rightarrow$ *psbe, FP* $\rightarrow$ *fp* $\}$

$$P_2 = \left\{ \begin{array}{ll} \textit{SCHED-PS-PS} \rightarrow \textit{PS}, & \textit{SCHED-PS-PSBE} \rightarrow \textit{PS}, \\ \textit{SCHED-PS-ALL} \rightarrow \textit{PS}, & \textit{SCHED-PSBE-PS} \rightarrow \textit{PSBE}, \\ \textit{SCHED-PSBE-PSBE} \rightarrow \textit{PSBE}, & \textit{SCHED-PSBE-ALL} \rightarrow \textit{PSBE}, \\ \textit{SCHED-RESBS-RESBS} \rightarrow \textit{RESBS}, & \textit{SCHED-RESBS-PS} \rightarrow \textit{RESBS}, \\ \textit{SCHED-RESBS-PSBE} \rightarrow \textit{RESBS}, & \textit{SCHED-RESBS-ALL} \rightarrow \textit{RESBS}, \\ \textit{SCHED-TS-PSBE} \rightarrow \textit{NULL}, & \textit{SCHED-TS-RESBS} \rightarrow \textit{NULL}, \\ \textit{SCHED-TS-NULL} \rightarrow \textit{NULL}, & \textit{SCHED-TS-ALL} \rightarrow \textit{NULL}, \\ \textit{SCHED-TS-FP} \rightarrow \textit{NULL}, & \textit{SCHED-TS-PS} \rightarrow \textit{NULL}, \\ \textit{SCHED-LINUX-PSBE} \rightarrow \textit{FP}, & \textit{SCHED-LINUX-RESBS} \rightarrow \textit{FP}, \\ \textit{SCHED-LINUX-ALL} \rightarrow \textit{FP}, & \textit{SCHED-LINUX-FP} \rightarrow \textit{FP}, \\ \textit{SCHED-LINUX-PS} \rightarrow \textit{FP}, & \textit{SCHED-FP-PSBE} \rightarrow \textit{FP}, \\ \textit{SCHED-FP-RESBS} \rightarrow \textit{FP}, & \textit{SCHED-FP-ALL} \rightarrow \textit{FP}, \\ \textit{SCHED-FP-PS} \rightarrow \textit{FP}, & \textit{SCHED-FP-FP} \rightarrow \textit{FP} \end{array} \right\}$$

$$P_3 = \left\{ \begin{array}{ll} \textit{ALL} \rightarrow \textit{SCHED-FP-ALL}, & \textit{ALL} \rightarrow \textit{SCHED-LINUX-ALL}, \\ \textit{ALL} \rightarrow \textit{SCHED-PS-ALL}, & \textit{ALL} \rightarrow \textit{SCHED-PSBE-ALL}, \\ \textit{ALL} \rightarrow \textit{SCHED-TS-ALL}, & \textit{ALL} \rightarrow \textit{SCHED-RESBS-ALL}, \\ \textit{RESBS} \rightarrow \textit{SCHED-FP-RESBS}, & \textit{RESBS} \rightarrow \textit{SCHED-LINUX-RESBS}, \\ \textit{RESBS} \rightarrow \textit{SCHED-RESBS-RESBS}, & \textit{RESBS} \rightarrow \textit{SCHED-TS-RESBS}, \\ \textit{PS} \rightarrow \textit{SCHED-FP-PS}, & \textit{PS} \rightarrow \textit{SCHED-LINUX-PS}, \\ \textit{PS} \rightarrow \textit{SCHED-PS-PS}, & \textit{PS} \rightarrow \textit{SCHED-PSBE-PS}, \\ \textit{PS} \rightarrow \textit{SCHED-RESBS-PS}, & \textit{PS} \rightarrow \textit{SCHED-TS-PS}, \\ \textit{PSBE} \rightarrow \textit{SCHED-FP-PSBE}, & \textit{PSBE} \rightarrow \textit{SCHED-LINUX-PSBE}, \\ \textit{PSBE} \rightarrow \textit{SCHED-PS-PSBE}, & \textit{PSBE} \rightarrow \textit{SCHED-PSBE-PSBE}, \\ \textit{PSBE} \rightarrow \textit{SCHED-RESBS-PSBE}, & \textit{PSBE} \rightarrow \textit{SCHED-TS-PSBE}, \\ \textit{NULL} \rightarrow \textit{SCHED-TS-NULL}, & \textit{FP} \rightarrow \textit{SCHED-FP-FP}, \\ \textit{FP} \rightarrow \textit{SCHED-TS-FP}, & \textit{FP} \rightarrow \textit{SCHED-LINUX-FP} \end{array} \right\}$$

$$P_4 = \left\{ \begin{array}{lll} \textit{RESBS} \rightarrow \textit{RESBS RESBS}, & \textit{PS} \rightarrow \textit{PS PS}, & \textit{PSBE} \rightarrow \textit{PSBE PSBE}, \\ \textit{NULL} \rightarrow \textit{NULL NULL}, & \textit{FP} \rightarrow \textit{FP FP} & \end{array} \right\}$$

2.3.3 Algorithm for the word-problem / Hierarchy Generation

For context free grammars polynomial algorithms exist that solve the word problem, considering the possibility that for each guarantee word more than one syntax tree might exists (such a grammar is called ambiguous) the well known CYK - Algorithm cannot be applied to this grammar. Instead we will use the $\Theta(n^3)$-Parsing Algorithm presented by Earley.

The following algorithm allows the generation of the corresponding scheduler hierarchy for a given generated syntax tree.

Definition 14 *Given: a node r, root of a subtree t , denoted by its description d(r) and its children c(r), ordered from left to the right and the guarantee word w = $g_1\ g_2, \ldots\ g_n$*
Output: G, a set of guarantees, representing the subhierarchy generated from t

Algorithm BUILD HIERARCHY (Root r):
let w be $g_i \ldots g_n$ with $1 \leq i \leq n$

1. *if d(r) is a terminal → set w = $g_{i+1} \ldots g_n$ and return $\{g_i\}$*

2. *if $\exists$ a scheduler type ρ and a guarantee type σ, such that $d(r) = SCHED\_\rho\_\sigma$*
   $\rightarrow$ create scheduler s of type ρ and create a incoming guarantee g of type σ for s
   let $G' = \emptyset$
   $\forall c \in c(r)$: $G' = G' \cup$ BUILD HIERARCHY(c)
   $\forall g' \in G'$: add g' as contract to s
   return $\{g\}$

3. *else:*
   let $G' = \emptyset$
   $\forall c \in c(r)$: $G' = G' \cup$ BUILD HIERARCHY(c)
   return G'

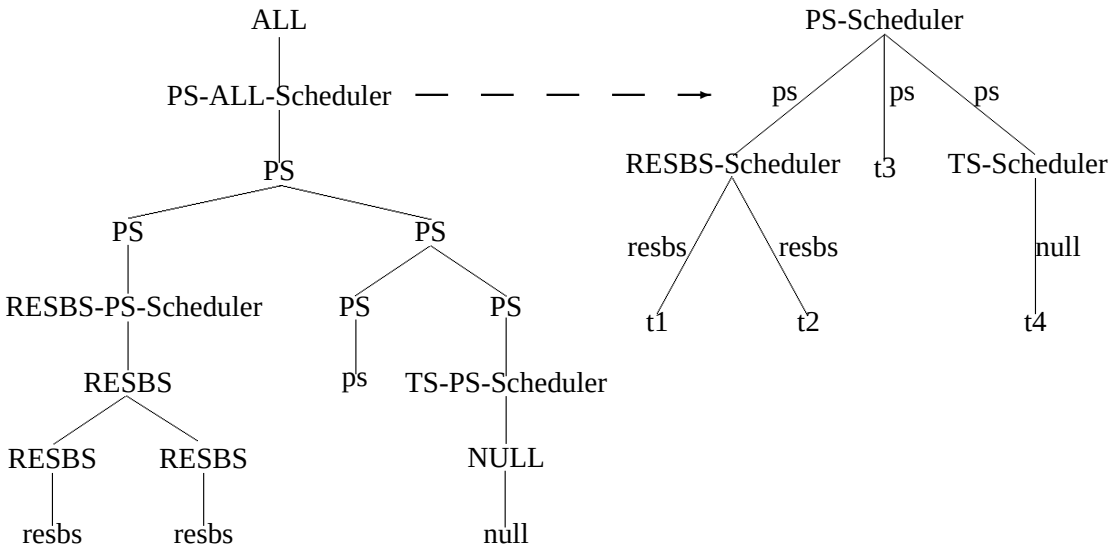
Here a short explanation of the different algorithm steps:

- First case: The algorithm has found a terminal symbol. It can now choose the next guarantee from the guarantee word, due the fact that we visit both syntax tree and guarantee word from left to right.

- Second case: BUILD HIERARCHY has found a ”scheduler production”, this means a scheduler has to be created, with the corresponding ingoing guarantee, since the defined context free grammar does not include direct guarantee conversions, the guarantee can be returned without conversion.

- Third case: The algorithm has found a production $guarantee \rightarrow guarantee\ guarantee$ to model multiple guarantees below a certain scheduler, the algorithm just needs to return the recursive results of all subtrees.

Figure 2.4: An exemplary syntax tree and the derived hierarchy



For a given set of application threads $t_1, \ldots, t_4$ a corresponding guarantee tuple $\{resbs, resbs, ps, null\}$ has been generated, figure 2.4 shows a possible syntax tree and the generated *meta hierarchical scheduler*.

Observations have shown that the by the randomized Earley Algorithm generated syntax trees and corresponding hierarchies have several disadvantages concerning the number of schedulers and the hierarchies depth. For this the following trivial minimizing operations have been implemented:

Definition 15 *Given a scheduler s, root of a subhierarchy and its guarantee contracts G(s), each g ∈ G(s) has an owner o(g), which is child of s.*

MINIMIZATION OPERATIONS for a root r:

1. $\exists g \in G(s)$: *o(g) is a scheduler and o(g) and s are of the same scheduler type*
   $\rightarrow$ *move all contracts G(o(g))from o(g) to s, remove o(g) from the hierarchy*

2. $\exists g \in G(s)$: *o(g) is a scheduler and* $\exists g_1 \in G(s), g_2 \in G(o(g))$: g_1 *and* g_2 *are of the same type*
   $\rightarrow$ *move* g_2 *from o(g) to s and remove o(g) from the hierarchy, if G(o(g)) =* $\emptyset$

3. $\exists g_1, g_2 \in G(s)$: $o(g_1)$,$o(g_2)$ *are schedulers and are of the same type*
   $\rightarrow$ *move all contracts G(o(*g_2*))from o(*g_2*) to o(*g_1*), remove o(*g_2*) from the hierarchy*

4. $\exists g_1, g_2 \in G(s)$: $o(g_1)$,$o(g_2)$ *are schedulers and* $\exists g_1^{'} \in G(o(g_1)), g_2^{'} \in G(o(g_2))$: $g_1^{'}$ *and* $g_2^{'}$ *are of the same type*
   $\rightarrow$ *move* $g_2^{'}$ *from o(*g_2*) to o(*g_1*) and remove o(*g_2*) from the hierarchy, if G(o(*g_2*)) =* $\emptyset$

The first two operations will reduce the hierarchies depth, the third and the fourth examine neighboring scheduler and reduce the hierarchy width. A hierarchy is called minimized, if and only if for each scheduler in the hierarchy no minimization operation condition is fulfilled and no operation can be used. The average depth and number of schedulers for a randomized guarantee word g with $|g|$ = 40 is three and eleven scheduler respectively.

2.4 Fitness and Genetic Operators

by Simon Muras, Mattias Stöneberg

Generating hierarchies from given application requirements and representing them by a tuple of guarantees does not guarantee, that the generated hierarchy is the most appropriate *meta hierarchical scheduler*, we even can not determine, what *appropriate* means in this context.

To ensure that at least almost optimum results can be guaranteed we‘ll first introduce a short definition of a composed fitness function, followed by the description of genetic strategies, which will help us, finding the optimum in the given search space (=set of legal MHS).

For INTERACTIVE, MULTIMEDIA(which means Soft/Realtime Threads) or BATCH threads (these are the major thread classes) optimization criterias are *response time, deadline failure rate* and *turnaround time* (for a definition of these criteria see chapter Testing). Also for each thread the importance value has been defined (the lower the importance value ($\geq$1), the more important it becomes).

Due to the fact, that a MHS‘s goodness depends on the target architecture, theoretical evaluation of the hierarchy would be incongruous to regain an adequate fitness value, so precise testing is essential.

We will now introduce a basic approach on how the evaluation of an atomic fitness function for threads will lead us to the definition of a composed fitness function for *meta hierarchical scheduler*:

Definition 16 *If opt(t) is the defined optimum value of a given Thread t(e.g. Response Time of 20 ms) and test(t) the measured value, imp(t) the importance of a thread t ∈ T, and let* $max = max\{imp(t)|t \in T\}$ *and* $min = min\{imp(t)|t \in T\}$*. Then the fitness* fitness(t) *of t can be defined as:*

$$fitness(t) = \frac{max + min - imp(t)}{max} * \begin{cases} opt(t) < test(t): & \frac{opt(t)}{test(t)} \\ else: & 1 \end{cases}$$

We are now able to define this composed fitness function.

Definition 17 *If fitness(i) with i* $\epsilon \mathbb{R}$ *is the fitness function with i instead of test(t), then the fitness* fitness(H) *of a MHS H with its Threads T is defined as fallow:*

$$fitness(H) = \frac{\sum_{t \epsilon T} fitness(t)}{\sum_{t \epsilon T} \frac{max+min-imp(t)}{max}}$$

This means the fitness function is a quotient of the summed test results and optima.

Due the fact that the response time's unit is ms, the turnaround time's is sec and the deadline failure rate is $\in \mathbb{N}$, the most complex task dealing with fitnesses of hierarchies is the normalization of measured values.

2.4.1 Integration with GAs

We will now discuss the problem of defining an adequate genetic strategy. Genetic Algorithms use the operations *mutation, recombination* and *selection* to find the *optimum* of a given *fitness* function *f* on a set of individuals called *population*. For a given search domain H and two individuals h_1, $h_2 \in H$ with a distance function d(h_1, h_2) the following guidelines for mutation and selection probabilities P should be fulfilled.

Definition 18 *For a given search space* H *and two individuals* h_1, $h_2 \in H$ *with a distance function* $d(h_1, h_2)$ *the following guidelines for mutation and recombination probabilities* P *should be fulfilled.*

- $\forall h_1, h_2 \in H : P(mutation(h_1) = h_2) > 0$
- $\forall h_1, h_2, h_3 \in H :$
  $d(h_1, h_2) < d(h_1, h_3) \implies P(mutation(h_1) = h_2) > P(mutation(h_1) = h_3)$
- $\forall h_1, h_2, h_3 \in H :$
  $d(h_1, h_2) = d(h_1, h_3) \implies P(mutation(h_1) = h_2) = P(mutation(h_1) = h_3)$
- $\forall h_1, h_2, h_3 \in H$ with $h_3 = recombination(h_1, h_2) :$
  $max\{d(h_1, h_3), d(h_2, h_3)\} \leq d(h_1, h_2)$
- $\forall h_1, h_2 \in H :$
  $P(d(h_1, recombination(h_1, h_2)) = \alpha) = P(d(h_2, recombination(h_1, h_2)) = \alpha)$

By regarding the set of valid hierarchies as the search domain H the problem of defining a metrics (with symmetry and triangle inequality as basic criteria) the guidelines defined in 18 becomes exceedingly complicated.

Instead we will define recombination and mutation on guarantee tuples, defined in 2.1.1 and presented in 2.3.1. For a set of threads a number of distinct guarantee tuples G can be formulated, taking randomly a subset G' of G as basic population, and defining the operations $mutation : G \rightarrow G$ and $recombination : G \times G \rightarrow G$. With a certain *mutation* probability p (e.g. $\frac{1}{n}$ with $g \in G : g = g_1 g_2 \dots g_n$) we flip guarantee g_i, $i \in \{1 \dots n\}$ into a guarantee h and construct the new guarantee word $g' = g_1 \dots g_{i-1} h g_{i+1} \dots g_n$.
Assuming that the guarantees in each guarantee word are ordered by their threads even recombination can be described, given $g, h \in G : g = g_1 g_2 \dots g_n \; h = h_1 h_2 \dots h_n$ and with a certain *recombination* probability p' a position i, such that after recombination the guarantee words $g', h' \in G : g' = g_1 \dots g_i h_{i+1} \dots h_n$, $h' = h_1 \dots h_i g_{i+1} \dots g_i$ are created.
Selection can be realized by the technique of *stochastic sampling with replacement*, means a guarantee word $g \in G'$ where $G' = \{g_1, \dots, g_m\} \subseteq G$ will be part of the next population G'' with at least the probability $\frac{fitness(g)}{\sum_{i=1}^{m} fitness(g_i)}$.
An appropriate adjustments of *mutation probability, recombination probability, population size, number of generations* can be found in the results section.

3 SDL

by Janni

In order to ensure a high diversity of system characterizations from facile descriptions (e.g. made by a novice user) to very detailed and well analyzed platform specifications with restricted number of runnable processes we developed n unique system and application description language. Consequently a SDL-parser which decides the well known word problem for this language has been developed. First of all this parser reads in an SDL File, parses it and then generates a hierarchy of objects which are organized under a central parser object. Example objects are $Hardware\ Requirements$ and $Application\ Requirements$.

The 'Scheduler Description Language' SDL is part of XML, well described by a complex taxonomy. A word $w \in SDL$, and thus a system's description basically consists of of these two components:

- Hardware Requirements
- Software Requirements

3.0.2 Hardware Requirements

This fragment of w describes the given architecture and processor specifications.

- Architecture is a simple parameter specifying the name of the considered architecture.
- The Processor element has multiple parameters which are:
  1. The name of the CPU - e.g. Athlon, PentiumII or such
  2. Cachesize - the size of the second-level cache
  3. Frequency - the frequency of the CPU in MHZ.
  4. Bus Speed - the speed of the memory bus.
- The memory attribute has the following options:
  1. Memory latency - the latency of the memory in msecs.
  2. Memory Speed - the speed of the memory in MHZ
  3. Memory Bandwidth - the bandwith of the memory in MBytes/sec

```
<HARDWARE_REQUIREMENTS>
      <ARCHITECTURE name="--">
          <PROCESSOR frequency="--" name="--" cachesize="--"
            dma="--" bus_speed="--" io_bus_speed="--"
```

```
            memory_bandwidth="--" memory_latency="-"
            memory_speed="--" number_of_cpus="--">
              <POWER_MANAGEMENT granularity="--"/>
          </PROCESSOR>
      </ARCHITECTURE>
  </HARDWARE_REQUIREMENTS>
```

3.0.3 Application Requirements

Application behaviour and its threads can be formulated by this section of the SDL. Each application basically can be described by the fallowing two attributes: name of the application and it's path.

```
<APPLICATION_REQUIREMENTS>
  <APPLICATION name="emacs" path="/usr/sbin/">
</APPLICATION_REQUIREMENTS>
```

The threads of an application at this point of implementation could be divided in three disjoint sets:

1. Interactive-Threads
2. Batch-Threads
3. Periodic-Threads

Each thread can also be associated with a set of attributes which are the following:

1. ID - for a unique identifier inside the operating system
2. Importance
   The importance is a number between 1 and ∞. The lower the importance value, the more important a thread becomes. High importance means a higher allocation of free resources and higher influence on the fitness of hierarchies.

The attributes above can be applied to any type of thread while the fallowing attributes can only be applied to threads of certain type.

Periodic-Threads have the following extra attributes:

1. Amount, the average amount of cpu-time in ms per period
2. Amount Unit
   can be microsecsonds (usec), miliseconds (ms) or seconds (s)
3. Period, e.g. a amount of $20\ ms$ in a period of $100\ ms$
4. Period Unit
   can be usec, ms or s
5. Deadline Failure
   An optimum value can be set for each thread $\Leftrightarrow$. Its value is $i \in \mathbb{N}$ and 1 by default.

```
<APPLICATION_REQUIREMENTS>
  <APPLICATION name="--" mode="--">
  <PERIODIC id="12" importance="2" amount="12"
    period="13" amount_unit="ms" period_unit="ms"
    deadline_failure_rate="1"/>
</APPLICATION_REQUIREMENTS>
```

Batch-Threads have the following extra attributes:

1. Share, the average share of cpu-time in %
2. Turnaroundtime
   An optimum value can be set for each thread.
3. Turnaroundtime Unit
   can be usec, ms or s

```
<APPLICATION_REQUIREMENTS>
  <APPLICATION name="--" mode="--">
  <BATCH id="3" importance="2" share="1"
         turnaround_time="10"\\
   turnaround_time_unit="ms"/>
</APPLICATION_REQUIREMENTS>
```

Interactive-Threads attributes:

1. Response Time
   An optimum value can be set for each thread.
2. Response Time Unit
   can be usec, ms or s

```
<APPLICATION_REQUIREMENTS>
  <APPLICATION name="--" mode="--">
  <INTERACTIVE id="10" importance="1" response_time="10"
               response_time_unit="ms"/>
</APPLICATION_REQUIREMENTS>
```

After creating the object hierarchy this structure is used by the GUI in order to visually analyse the given SDL. The hierarchy also is used by the ASM-Generator in order to generate an adequate MHS (Meta Hierarchical Scheduler).

4 Testing SAADI

by Tobias Malbrecht

This chapter deals with testing SAADI. More precisely, testing SAADI means the testing of hierarchical schedulers generated with the SAADI framework and afterwards included into a SAADI enhanced Linux kernel. Testing SAADI is necessary for two reasons: (1) to compare the performance of SAADI schedulers with other existing schedulers (e.g. the $O(1)$ scheduler by Ingo Molnar, which is part of the current Linux 2.5 development kernel), (2) to evaluate different scheduler hierarchies. Since we have not spent any time on the first issue yet, we concentrate on the second intention of testing SAADI.
Section 4.1 introduces the testing processes with the aim to find an optimal scheduler hierarchy. Section 4.2 gives a little overview of what parameter we want to measure when testing a SAADI kernel for evaluation of scheduler hierarchies. In section 4.3 we present the tools we use for testing. The topic of section 4.4 is simulation of real applications by synthetic ones. In section 4.5 we describe the testing framework in detail.

4.1 Testing cycle of SAADI

As outlined in chapter 2.4 SAADI pursues an evolutionary approach when generating scheduler hierarchies. This necessitates the computation of a fitness value for each generated hierarchy. This is a complex task, since the only way of computing the fitness of a scheduler hierarchy is to run the generated scheduler code representing this certain hierarchy on a real operating system and to measure these values that have influence on the fitness of the underlying scheduler hierarchy. The main complication arises from the fact that the evolutionary algorithm itself and the computation must be split into two different phases. This is however based on the fact, that we designed the high level part of SAADI on one side (i.e. first of all the evolutionary algorithm to find an optimal scheduler hierarchy) as a Java program. On the other side the computation or, to be more precise, the evalution of scheduling behaviour has to be done by running a real operating system (i.e. a SAADI Linux kernel with a scheduler corresponding to a certain hierarchy), without a chance to directly interact with the Java part of SAADI.
The scenario described above leads to the testing cycle as shown in figure 4.1. The whole procedure is coordinated by shell scripts. In the following we describe the components of the testing cycle that help in understanding the whole testing process. At first we will explain parameters that help in the evaluation of a scheduler hierarchy.

4.2 Valuation of scheduler hierarchies

To determine the fitness of a generated scheduler hierarchy we have to measure a certain set of values. These values are scheduling criteria (from the point of view of a particular process):

- turnaround time

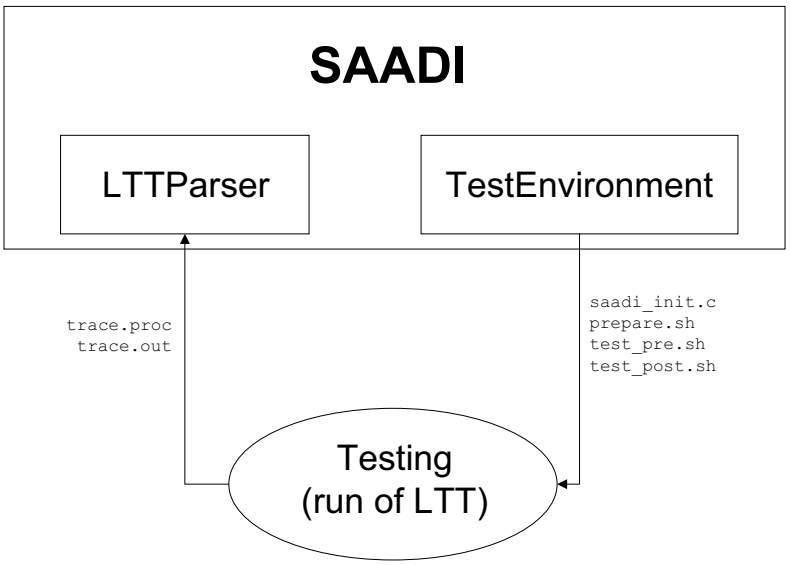


Figure 4.1: Testing cycle of SAADI

- response time
- deadline failure rate

In this regard it certainly does not make any sense to measure and to consider all these three values for any kind of process. One has to distinguish between three different types of processes (as specified in the SAADI description language): batch, interactive and real-time. Therefore we measure and want to optimize turnaround time for batch, response time for interactive and deadline failure rate for real-time processes[1]. If one measures those values for the processes that are announced in the SDL file for which a certain scheduler hierarchy is built, one can compute the fitness of that hierarchy from that (see chapter 2.4). The evolutionary algorithm used with SAADI then is able to find an optimal scheduler hierarchy for that certain set of applications specified in the SDL file.
The complication in the determination of this optimal hierarchy is therefor based on the subjects to measure the scheduling criteria mentioned above to evaluate hierarchies by computing their fitness from the measured values.

4.3 Measuring scheduling criteria in SAADI

As mentioned in the last section we need to perform a phase of testing and measurement while running a SAADI kernel to evaluate the underlying scheduler hierarchy. This evaluation is done by following

[1]Since we only consider basic soft real-time processes in this stage of development, we in fact do not measure deadline failure rate, but the grade of violation of RESBS-guarantees (i.e. number of times a process does not receive the guaranteed amount of cpu time during its period).

tools and components of SAADI that we present in the following.

4.3.1 Linux Trace Toolkit

Since it builds the basis of our measurement, we first concentrate on a brief description of the Linux Trace Toolkit (LTT). LTT is a kernel-level event logging system. A LTT enhanced linux kernel gains the facility to trace events at a number of points in the kernel code. Examples for these points are the entry and exit of IRQs or syscalls as well as a sched change, a process fork or wakeup. All these events are logged into a file (producing a relatively small overhead of less than $2,5\%$). Additionally a file is created before the start of the tracing, listing all existing processes and their names.

```
tracedaemon -ts60 /dev/tracer trace.trace trace.proc
```

will run tracing for 60 seconds and store the trace and the process file. To achieve an event file in a human (and SAADI) readable format we have to run

```
tracevisualizer trace.trace trace.proc trace.out
```

subsequently. Figure 4.2 examplarily shows a part of a trace file created that way. As one could see,

```
File system          1,044,463,092,373,065    421     20      POLL : 7; TIMEOUT : 0
Memory               1,044,463,092,373,066    421     12      PAGE ALLOC ORDER : 0
Timer                1,044,463,092,373,068    421     17      SET TIMEOUT : 3501
Sched change         1,044,463,092,373,070    0       19      IN : 0; OUT : 421; STATE : 1
IRQ entry            1,044,463,092,377,089    0       9       IRQ : 10, IN-KERNEL
IRQ exit             1,044,463,092,377,099    0       7
Soft IRQ             1,044,463,092,377,101    0       12      SOFT IRQ : 2
Network              1,044,463,092,377,102    0       12      PACKET IN; PROTOCOL : 8
Process              1,044,463,092,377,123    0       16      WAKEUP PID : 394; STATE : 1
Sched change         1,044,463,092,377,129    394     19      IN : 394; OUT : 0; STATE : 0
File system          1,044,463,092,377,132    394     20      SELECT : 4; TIMEOUT : 2147483647
```

Figure 4.2: Sample trace

this example trace contains two of the events, which are relevant when testing SAADI. The first one is a process wakeup in line 9, when the process with PID 394 moves from ready queue to run queue. The second one is the following sched change from the process with PID 0 (i.e. the idle task) to the process with PID 394 which was just woke up before. For the sake of completeness we have to mention that there is another sched change already in line 4 due to a timer timeout invoked by the process with PID 421.

4.3.2 Preparing the testphase

For setting up the testing environment SAADI includes a module named TestEnvironment. This module provides two functions: (1) it generates the scheduler code for the scheduler hierarchy generated by the SAADI code generator, (2) it generates scripts, which prepare and run our test applications. After having created a subdirectory for the actual hierarchy (with its ID as name), it creates the scheduler initialisation code (`saadi_init.c` and the scripts `prepare.sh`, `test_pre.sh` and `test_post.sh` in this subdirectory. `prepare.sh` copies the applications to the same hierarchy subdirectory. The two test scripts `test_pre.sh` and `test_post.sh` start the applications which should run to test the hierarchy. `test_pre.sh` starts interactive and soft real-time processes, which should run for the whole LTT run, and is therefore started before the LTT. `test_post.sh` starts all batch applications and is therefore started immediately after the LTT is started.

4.3.3 LTTParser

The LTTParser is actually the interface between the testing framework and the evolutionary algorithm. To calculate turnaround time, response time and guarantee violations SAADI parses the output generated from LTT. The LTTParser included in the SAADI.sim package searches for relevant events which are: process fork, process termination, sched change, IRQ entry and custom SAADI events. Internally the LTTParser creates a hash table of ProcessStatistic objects (one for each process appearing in the trace) in which the exact forking or termination times of a process, as well as the times a process got or lost the CPU, are stored. Then the turnaround time of a process can be calculated as follows[2]:

$$t_{turnaround} = t_{termination} - t_{fork} \tag{4.1}$$

The response time of a process (i.e. the average response time) is measured by considering the appearance of a special IRQ entry event and a following [3] sched change (n is the number of sequences of the IRQ entry event and a following sched change to that process occuring in the trace):

$$t_{response\ time} = \frac{\sum_{k=1}^{n} t_{IRQ\ entry_k} - t_{sched\ change_k}}{n} \tag{4.2}$$

The number of deadline misses or the number of RESBS-guarantee violations is the number of times the following inequation is true when going over every period from trace start until trace end time:

$$t_{guaranteed\ amount} > t_{measured\ amount} \tag{4.3}$$

After having analyzed the trace the ProcessStatistic objects are attached to the given hierarchy (which is the one the tested kernel was generated from).

4.4 Simulation of applications

While testing SAADI hierarchies (by running LTT), we have to simulate applications, which are specified in an SDL file. The disadvantage of performing the tests with real applications compared to synthetical applications is that we can not describe the requirements of real applications as exact as we need to get exact results from testing. Instead of real we therefore use synthetical applications, whose requirements can be predetermined and whose behaviour is predictable or even controllable. Because of the different requirements in consideration of using the event based LTT one has again to differentiate between batch, interactive and soft real-time processes.

Batch processes

Intending to simulate batch processes, we developed a tool named SAADI_bpg. This program does not represent a batch process itself, but generates programs, that represent batch processes with a desired execution time. By calling

```
SAADI_bpg -n test.c -t 20000
```

[2] Every time value mentioned in the following euquations relates to the corresponding process.

[3] Unfortunately the current SAADI testing environment does not consider, that the following sched change may not grant control to the program, which should be affected by the IRQ, but rather to another one. To deal with that problem, future work may be done to establish an accurate connection between IRQ events and the appropriate sched changes they provoke.

the tool generates a C program named `test.c`, which after being compiled represents a batch process. This batch process will run for about 20 seconds on an otherwise idle machine, which means, that it has an execution time for about 20 seconds.
SAADI_bpg has to be run under an otherwise slightly used or even idle system, because it measures the time it takes to perform a certain action (e.g. a matrix multiplication). It then computes how often this action can be performed to almost reach exactly the desired execution time of the batch process which is to be generated. It is obvious that under the condition of a slightly loaded system, the turnaround time of a process is approximately equal to its actual execution time. Therefore the execution time of the generated process is the optimum of turnaround time of that process in general. If that process is then run under a heavily loaded system (as specified by an SDL file), we would be able to measure the real turnaround time of that process (i.e. under real conditions) and to compute the divergence from its optimal value.
The generated batch programs include the facility to throw respectively one LTT user event at the start of running and one at the end. This is necessary due to pecularities of the LTT and substitutes the analysis of process fork and process termination events.

Interactive processes

Simulation of interactive processes may be the most complex task when replacing real applications by synthetical ones. Interactive means that we want to measure how much time it takes until a user interaction is processed. On the operating system layer we define this time as the difference between the time an IRQ entry occured and the time the process is scheduled.
Since it is very difficult and time-consuming to explore we did not concentrate on simulating real user interaction like keystrokes or pressing mouse buttons. Hence we emulated this user interaction by activity on a network. Therefore we use an application written by Bill Hartner (unfortunately no documentation except the readme file provided with this application was available). This application is a chat client and server workload benchmark and consists of a client and a server program. As simulation of our interactive processes we run the server program which receives packets over a certain network interface card with a fixed IRQ (by running the client program on another computer, sending those packets). Once a packet was received, a IRQ entry event was signaled and logged by the LTT. A sched change event to the server program is likely to follow. These two events build the bounds for measuring the response time.
It may be useful to repeat this act of measuring response time a few times during a run of the LTT and then take the average response time or (since we want to ensure that response time does not exceed a reasonable value) the maximum response time as basis for optimization.

Soft real-time processes

Basic soft reservation processes can be easily modeled with a tool named Hourglass. Hourglass is a synthetic application, which provides different thread execution models. So its general approach is to create threads, which runs any workload during the time the thread is specified to run. From the various thread execution models we only use periodic ones as representation of basic soft reservation processes.

4.5 Running the tests

Finally we will give a little overview of the mechanism that generally controls the testing. At the moment we use some shell scripts for that. The one that controls the overall process is named `test_all`. Once SAADI has been started with an SDL and created an SDL directory, it calls SAADI again to add new hierarchy test environments (see section 4.3.2). After that it calls the next script called `init_test` for one of these newly generated test environments which mainly compiles the SAADI kernel with the given scheduler code of a hierarchy. The computer then is rebooted and forced to start `run_test` afterwards. This script actually starts the testing procedure by running the scripts generated by the test environment: `prepare.sh` at first, second `test_pre.sh`. Then the LTT (the `tracedaemon`) is started as described in 4.3.1 and just after that, `test_post.sh`. If all has been successful, then the trace is transformed by calling the `tracevisualizer`. A locking mechanism prevents a testing environment from being tested twice because the `test_all` script chooses another unlocked test environment for the next test. If no more unlocked test environments exist, it again calls SAADI including its evolutionary algorithm to generate new hierarchies and test environments which will also be tested as described above.

5 Implementation

5.1 Java

5.1.1 The Abstract Scheduling Model (ASM)

by Mattias Stöneberg

This section deals with the package `edu.udo.saadi.asm` which is one of the basic parts of the SAADI framework. Purpose of this package is the generation of an Abstract Scheduler Model, that is an instance of a Scheduling Hierarchy. The Classes defined in this package utilize the SDL-parser which is defined in `edu.udo.saadi.parser` to create Java objects that reflect the user specification of the system environment and the software used with that system. Using these information the ASMGenerator class tries to generate a hierarchy which fulfills all the requirements of the specified software. The resulting Scheduling Model is then integrated into the Linux Kernel by the use of the CodeGenerator class.

Before having a look at the generation of a hierarchy and at the genetic process, a detailed description of the structures of a single hierarchy and its sub-components is necessary.

Hierarchy (`Hierarchy.java`)

The real structure of the hierarchy is a reference to its root scheduler. This reference is saved in the `root` attribute. It is a tree of scheduler objects with task objects as leaves. Each leaf represents a task specified by the user. All the other tasks not specified will be scheduled by the scheduler referenced by the `defaultScheduler` attribute.

Since hierarchies are considered as points in a search-space of a genetic process and since they can satisfy the requirements in a worse or better way, hence these hierarchies have to satisfy a compariosn metric. Comparison of hierarchies is done by the use of a goodness value. This goodness is a value in the range from 0 to 1 where 1 is the optimum. It is assigned to the hierarchy after its evaluation in a test run within the Linux kernel. The evaluation is done by the use of a corresponding statistic object which is stored in the attribute `statistic` and its set-method. A detailed description of fitness computation is given in 2.4.

To distinguish between several hierarchies each is given an unique ID which consists of several variables. The method `getID()` returns a String which has the following form:

“H”+(value of `id`)+“At”+(value of `creationDate`)+“Generation”+(value of `generation`).

Schedulable (`Schedulable.java`) In the scheduler hierarchy the objects to be scheduled are tasks as well as other schedulers. The SAADI framework therefore defines the `Schedulable` class. The Task and Scheduler class inherit from `Schedulable` which defines a parent schedulable object and an in-guarantee which is assigned during the generation process.

Scheduler (`Scheduler.java`) This is another parent class which defines the following structure :

All guarantees which a scheduler exports to schedulables are grouped in the `contracts` attribute. This defines a set of children scheduled by the scheduler which can be accessed by the method `getChildren()`. Children can be added to the set of contracts by `addContract()`. The contracts are an important part of a hierarchy, because they define the way cpu time is shared among the processes.

Until now, the following schedulers are implemented:

- The LINUXScheduler represents Ingo's $O(1)$ scheduler
- The FPScheduler realizes a scheduler that schedules tasks with fixed priorities between 0 and 139; it has a timeslice as attribute.
- The TSScheduler is a time sharing or round robin scheduler with a timeslice as attribute.
- The PSScheduler represents a scheduler for tasks that requires a certain proportional share of cpu time; there is a timeslice-attribute, too.
- The PSBEScheduler inherits from a PSScheduler but considers an additional bounded error for each share.
- The RESBSScheduler realizes a reservation basic soft scheduler that schedules tasks which need a certain amount of time in a certain period of time;
- The RESPSScheduler represents a reservation probabilistic scheduler.

Task (`Task.java`) In addition to the attributes inherited from `Schedulable.java` an instance of this class holds a process statistic (`ProcessStatistic.java`) defined in the `sim` package of SAADI and a thread object `Thread.java`. The former one is only valid, if the hierarchy in which it is defined is already tested and the result of the test has been computed. The later attribute describes a 1-to-1 relation between a task object and a thread specified by the user in the SDL file.

Guarantees (`Guarantee.java`) After this description of schedulers and tasks which are the leaves of a scheduling hierarchy we want to outline the relationship between two nodes. As mentioned before a scheduler contains a set of contracts that each refer to a set of guarantees. A Guarantee consists of two important attributes: a scheduler object (`provider`) defines the guarantee and this guarantee is assigned to a schedulable object (`owner`) along with a `status`. The provider and owner objects refer to the parent and child relationship of a tree.

Different schedulers may provide different types of guarantees that form a set of guarantee types. These types are represented by various classes which all inherit from the basic class `Guarantee.java`. Until now, we have implemented the following guarantees in regard to all provided schedulers:

- fixed priority guarantee (`FPGuarantee.java`) which is provided by a fixed priority scheduler; its parameter is a priority (integer from 0 to 139 where 0 is the highest priority),
- proportional share guarantee (`PSGuarantee.java`) which is provided by a proportional share scheduler; its parameter is `proportion` that represents the required cpu time in %

- proportional share with bounded error (`PSBEGuarantee.java`) which is the same like PS-Guarantee with the additional parameter `alpha` for the bounded error.
- reservation basic soft guarantee (`RESBSGuarantee.java`) which is provided by a reservation basic soft scheduler; its parameters are `amount` and `period`,
- reservation probabilistic (`RESPSGuarantee.java`) which is the same like RESBSGuarantee with the additional parameter `maxAmount`,
- static guarantee (`StaticGuarantee.java`); the root scheduler of the whole hierarchy gets the ALL_GUARANTEE (represented by integer 1), that means a scheduler with this guarantee gets all cpu time. There is a second static guarantee, NULL_GUARANTEE, represented by 0; it is provided by a Linux scheduler or time sharing scheduler. A schedulable with this guarantee has no cpu time guaranteed.

Now we can show an UML representation of a hierarchy and its components (figure 5.1).

Generation process of an ASM

In the previous sections we displayed the structure of a hierarchy including the structure of schedulers, tasks and guarantees. Next we describe the procedure of creating such a structure from the information we get from the SDL specification.

To find an appropriate and "well performing" hierarchy which considers all requirements is a difficult task. It is divided into two independent tasks: a genetic algorithm, which tries to find an optimum among all tested hierarchies and a randomized algorithm which automatically creates legal hierarchies. A legal hierarchy is a hierarchy with no guarantee conflicts. The genetic algorithm is implemented in `GeneticClass.java` which is encapsulated in the `edu.udo.saadi.asm.earley`-package.

These classes are accessed by the SAADI main class only by the use of a method of the ASMGenerator, which reflects one of the interfaces of the ASM module:

Interfaces The main interface to access the ASM package is provided by `ASMGenerator.java`. It consists of only two public methods, with them first populations can be created or a previously generated one be evaluated usin an genetic algorithms.

- `public Population main(..., SoftwareConstraints sC, ...)`:

  This method creates a new population (`Population.java`) using the guarantee mapper (`GuaranteeMapper.java`) and the given software constraints. An instance of population contains a set of guarantee words (`GuaranteeWord.java`) which in turn contain a set of guarantees as well as a set of hierarchies. The main methods' task is to generate legal guarantee words for the starting population using the software constraints. This is done by the method `map(SoC,...)`: all guarantee words are equal in length, which is the number of tasks. They are generated in a certain way which consider the following condition: at any time the guarantees are fixed at one position for all guarantee words of any population.

  The purpose of the guarantee mapper (`GuaranteeMapper.java`) is to build a new population (`Population.java`) on the basis of the software constraints. An instance of the class `Population` contains a set of guarantee words (`GuaranteeWord.java`) which in turn

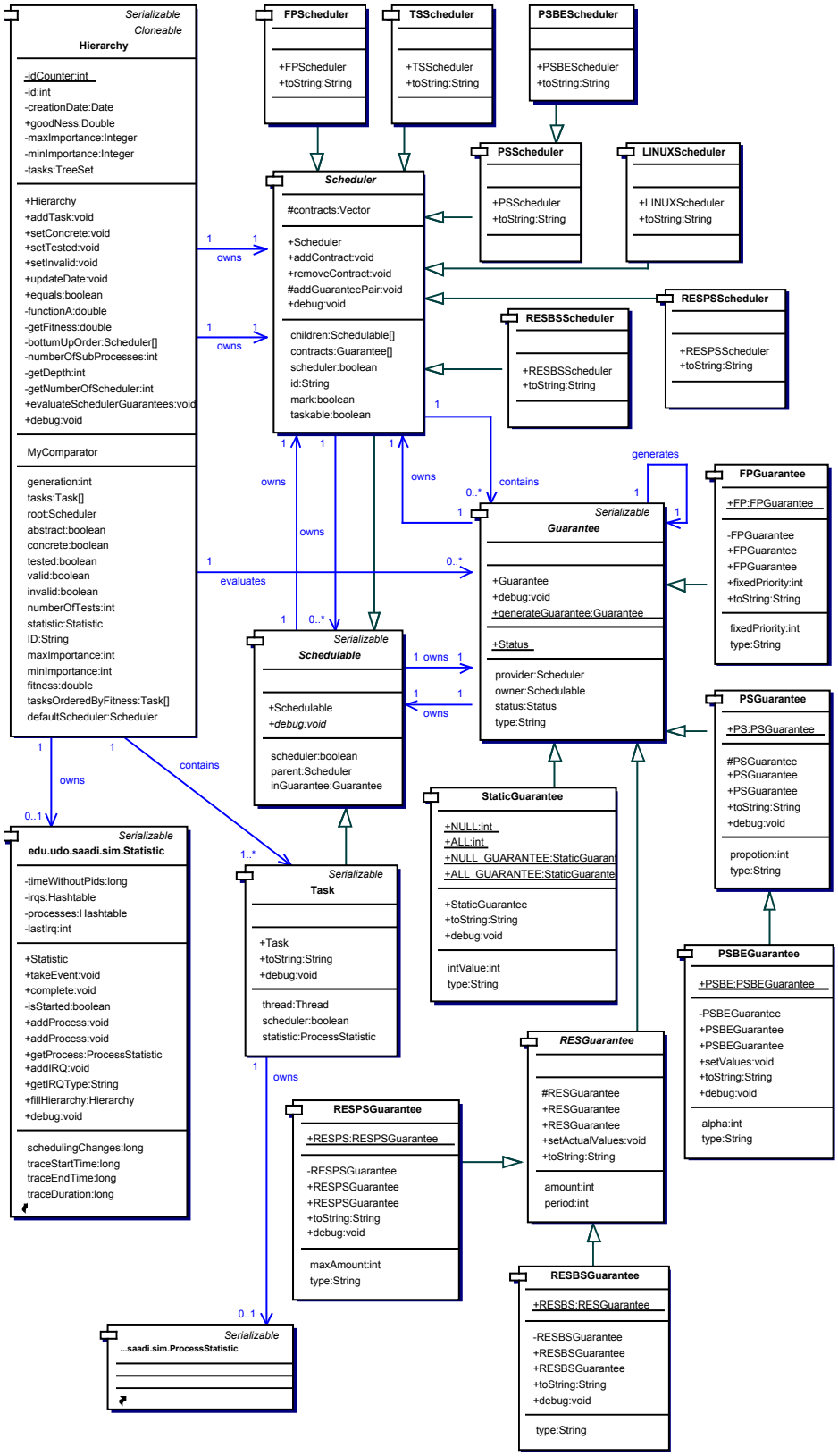


Figure 5.1: UML diagram of a hierarchy

contain a set of guarantees as well as a set of hierarchies. The method `main(..)` is responsible for creation of possible guarantee words reflecting the given software constraints. These words describe the starting population. All this is done in `main(SoC,...)`: All guarantee words are of the same length (length depends on the number of tasks). Every guarantee word fulfills the following condition:
The position of a guarantee in a guarantee word refers to the task to which this guarantee is related, thus a guarantee at position A of a guarantee word refers to the same task as a guarantee at position A of a different guarantee word (but of the same population). Different guarantees that meet the requirements of a task are evaluated. The resulting population contains all possible guarantee words (i.e. all combinations of guarantees) which the tasks can be connected to. A certain number of guarantee words is then used for further computation.

- `public Population main(Population pop)`:

  This method decides wether the population still contains hierarchies that are to be tested and if yes then it simply returns the given population and the next untested hierarchy, in case all hierarchies have been tested then it calls the genetic algorithm which generates a new population of hierarchies. A more detailed description of this process can be found in paragraphs below.

Another interface is defined by `Population.java`:

- `public void updateHierarchy(String id, Statistic stat)`:

  As the performance of hierarchies is measured using the test environment, therefore several statistics are collected. This method is used to set the statistics of a tested hierarchy in the population. The hierarchy is referenced by its ID. After a hierarchy has been tested then it is possible to compute the fitness value using the method `getFitness()` (see 2.4).

Figure 5.2 displays an UML representation of some classes from package `edu.udo.saadi.asm` as an overview for ASM generation.

Genetic Algorithm If the method `main(Population)` mentioned above is called with a population which only contains guarantee words with completely tested hierarchies then the method `computeNextGeneration(Population):Population` is called which continues the genetic algorithm of `GeneticClass.java`. Genetic algorithms depend on a stop criteria which is checked before the algorithm is continued. The stop criterium of our genetic process is represented by an instance of `StopCriterion.java`; which is fulfilled if the number of generations increases up to a former specified amount or a hierarchy of the population reaches a certain level of fitness. The method `stop(...)` is called to decide if the algorithm is to be continued or to be stopped. If it is to be continued the next generation of the population is computed as follows:

1. Selection (`select(Population):Population`):

   The process of "stochastic sampling with replacement" is used. This means that a guarantee word w_i will be taken over to the next generation with a probability of $\frac{fitness(w_i)}{\sum_j fitness(w_j)}$. The new population will be filled with words of the old one until it contains an equal amount of guarantee words.

2. Recombination (`recombine(GuaranteeWord[]):GuaranteeWord[]`):

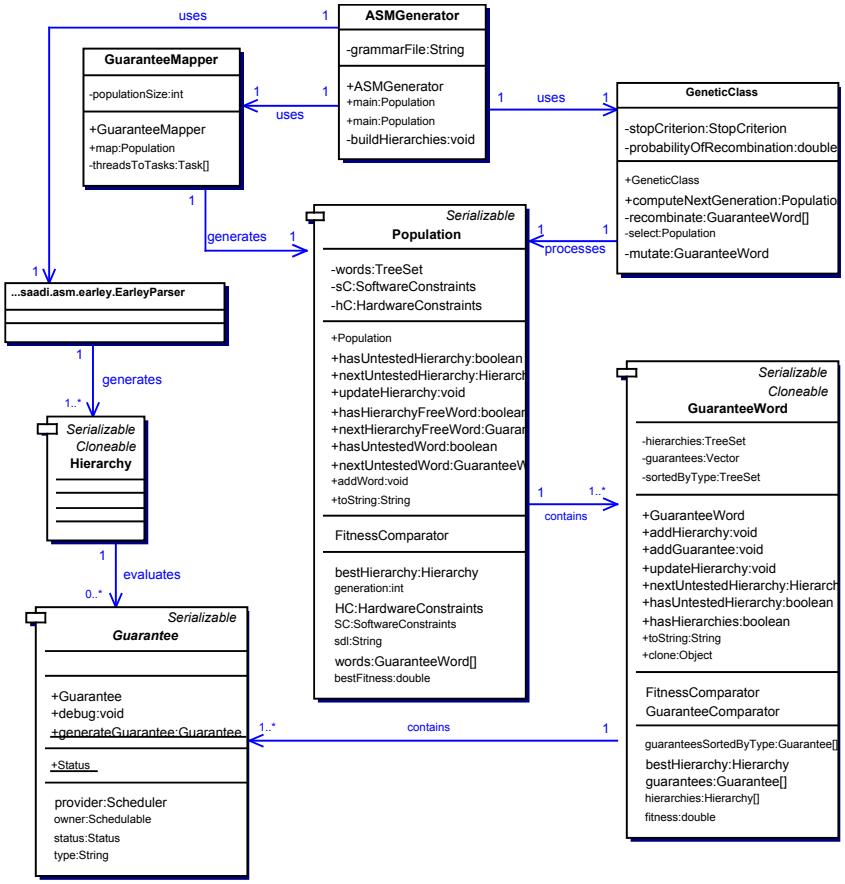


Figure 5.2: UML diagram for ASM generation.

The guarantee words of the old population are randomly put together in pairs and mixed with a given *probability of recombination* (given in the constructor) as follows:

An entry point to mix the words is chosen randomly and they are cut off to be exchanged and put together for a new word. As example $AAAA$ and $BBBB$ are cut off at a random point to A, AAA and B, BBB. Then they are put together to form the new words $ABBB$ and $BAAA$.

3. Mutation (`mutate(GuaranteeWord):GuaranteeWord`):

   Each guarantee word of the new population is mutated by exchanging the guarantee at each position of the word by a different guarantee at a probability of $\frac{1}{n}$ where n is the length of the word. The new guarantee has to be a legal one which does not replace the guarantee requirements.

With this procedure a new population is generated which contains new and modified guarantee words with a certain probability. These words neither contain untested nor already tested hierarchies.

This brings us to the step of "hierarchy generation".

Hierarchy generation A hierarchy will be generated for each word of a population until it is not already generated. To achieve this, `parseWord(GuaranteeWord,...):Hierarchy[]` of the class `EarleyParser.java` will be used. This class is the only interface to access the package `edu.udo.saadi.asm.earley`. The generation of hierarchies is divided into the following steps:

1. Earley Parsing: It is our aim to create syntax tress which can easily be converted into hierarchy by the use of a context free grammars and the word problem. In a context free grammar the rules of production refer to the relation of schedulers to schedulers and schedulers to tasks, represented by guarantee types. The grammar is an instance of `ContextFreeGrammar.java` which is parsed from the file `HL.cfg`. This file contains terminal symbols, non-terminal symbols, start symbols and rules of production. It is structured as:

   Each line contains a production rule in the form `A : B`, where as `A` is a terminal symbol and `B` is a non-terminal symbol. Terminal symbols refer to guarantees (of a guarantee word) connected to a task and are written lower case (e.g. `res`). Non-terminal symbols are written upper case and represent schedulers (`SCHED-INGUARANTEE-OUTGUARANTEE`) and the guarantees which are related to these schedulers (e.g. `RES`). As a scheduler may also have other schedulers as children a rule `A : A A` exists. The start symbol is the first symbol in the first line.

   Each guarantee word which provides the basis for construction of a guarantee consists of a set of guarantees. Considering the representation of guarantees as a String (`toString()`), a guarantee word represents a word in a context free grammar. By the use of the Earley Parser ([Earley, 1970]) any amount of randomly structured syntax trees can be extracted of such a word. To achieve this, the Earley Parser constructs a chart, which represents a certain type of Earley States (`EarleyState.java`) for every position in the word. After that the Viterbi Parser creates one or more syntax trees. The implementation relies on [Stolcke, 1995], section 3 and section 5.1 and works as following:

   The size of a chart is the size of a guarantee word plus one. After inserting a initial Earley State at position 0 the chart will be traversed (ascending, beginning with 0) and the method `operate(...)` called. This function is responsible for three tasks: call `scan(...)`, `complete(...)` and `predict()`. These methods test the former created chart and create an Earley State of type "scanned", "completed" or "predicted" if needed. For each position of the chart the search for a new Earley States is continued, until there are no additional

states after the three methods were called. There exists a final state at the last position of the chart, which should be reached in our case every time. Once final state is reached then the method `buildHiearchies(...):Hiearchy[]` can be used to construct one or more syntax trees. It uses the recursive function `buildParseTree(...)` which in turn basically implements the Viterbi Parsing algorithm. The only difference is the case in which a predecessor is to be chosen which is done in a random manner rather than the one with the best probability. The result consists of a tree of `ParseTree` objects which contain a string representing a symbol of the context free grammar and references to child nodes (`nodeLabel`). This tree constitutes a syntax tree.

2. Construction of a hierarchy: The next step in `buildHierarchies(...)` is to transform the constructed syntax trees into hierarchies (see 2.3.3). As the root scheduler has been set manually and all the tasks which refer to this guarantee word have been added to the hierarchy, the scheduler tree is created recursively by the use of `buildSubHierarchy(ParseTree, Scheduler):Guarantee[])`. If the parse tree contains a terminal symbols as label, the related guarantee of the guarantee word will be added to the scheduler as a contract, otherwise it may only be a symbol representing a scheduler or a guarantee between two schedulers:

   - In the case of schedulers an instance of the appropriate scheduler is created (by using the method `generateScheduler(String):Guarantee`) concerning the label of the parse tree object. This is done by extracting the type of scheduler and the in-guarantee from the symbol. The new scheduler object is added to the given scheduler by the in-guarantee. Afterwards all guarantees which are results of recursive calls to the method `buildSubHierarchy(...)` are added to the given scheduler.
   - If the guarantee exists between two schedulers it reflects a node which represents a guarantee in the hierarchy. Because of this only the guarantees of the siblings of the parse tree object are added (again by the use of `buildSubHiearchy(...)`).

   After a hierarchy is constructed, it reflects the syntax tree completely, but is quite big. Additionally the tasks are spread over the hierarchy inefficiently. Therefore a post processing is required to minimize the hierarchy. This is done in `minimizeHierarchy(...)` which primarily resolves chains like "PSScheduler → PSGuarantee → PSScheduler → PSGuarantee . . .". As this minimized hierarchy has its final form the guarantee parameters for all the guarantees between schedulers are missing. These are computed by `evaluateSchedulerGuarantees()` of `Hierarchy.java`. The method propagates the guarantee parameters which are already given (in the leaves, the parameters of the tasks) bottom-up-wise up until the root scheduler. Thus a schedulers' in guarantee is the sum of all the requirements of its siblings.

The number of constructed hierarchies per guarantee word is given as a parameter to the method `parseWord(...)` and can vary. After the generation phase is complete, the modified population containing new and/or old guarantee words is returned to the caller of `main(Population)`. If all hierarchies of a population are tested then a new generation is created.

Figure 5.3 displays an UML representation of the Earley Parser and its components.

5.1.2 The SAADISim(ulation) package

by Tobias Malbrecht

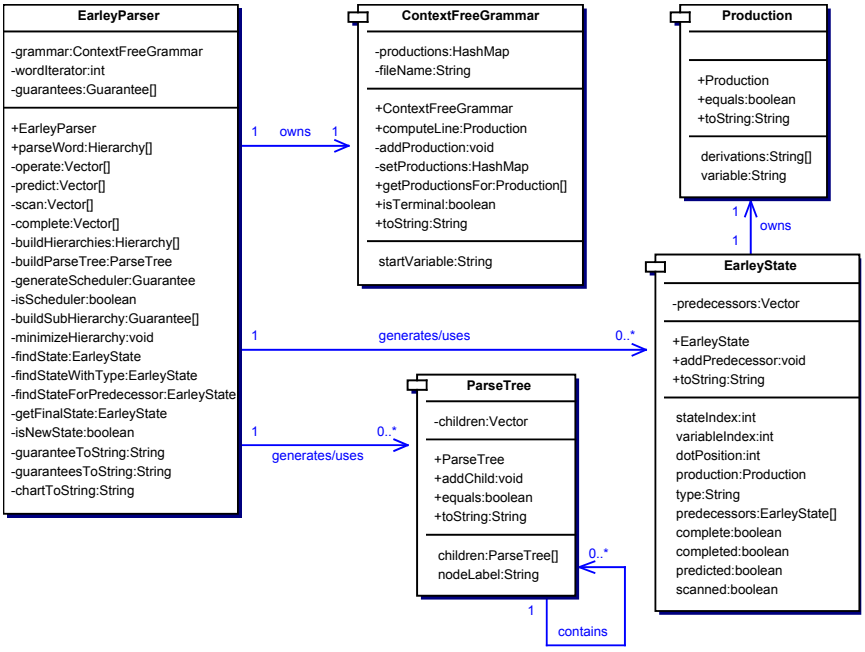


Figure 5.3: UML diagram of `edu.ud.saadi.asm.earley`.

In addition to the former described packages we developed another package which is called SAADISim (`edu.udo.saadi.sim`). This package is an important part of the scheduler hierarchies evaluation process, since it provides feedback to the ASMGenerator (described in section to facilitate optimization of the scheduler hierarchies. To get an overview of the testing and evaluation of scheduler hierarchies and the classification of SAADISim into this process we must refer to chapter. This section only provides an insight into the implementation of SAADISim.
To get a first overview of the classes provided with SAADISim see figure 5.4. In the following we describe the control flow of SAADISim.

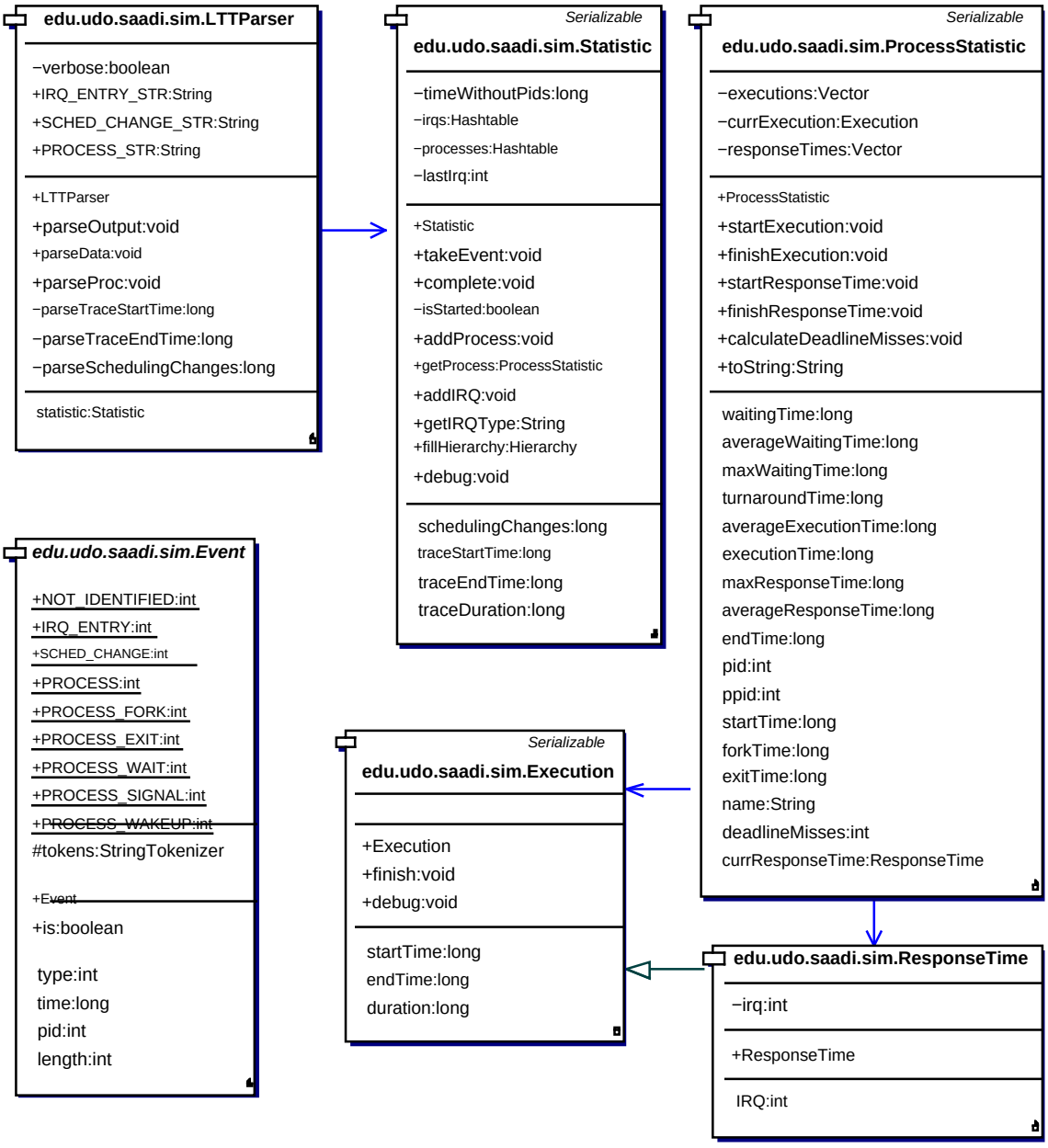


Figure 5.4: UML diagram of the data classes in the SAADIsim package

The class `LTTParser` parses the output files from the Linux Trace Toolkit (which is described in 4.3.1). It therefore creates an object of the class `Statistic` that represents the information extracted from Linux Trace Toolkit output in Java objects. During analysis, the `LTTParser` creates `Event` objects for every trace line of the LTT output that are relevant for the analysis of the scheduling behaviour and passes these events to the `Statistic` object which then examines and stores the information. Since there are several types of events, we implemented these events as subclasses of the class `Event`. Therefore the event received by the `Statistic` can either be an `EventIRQEntry`, an `EventProcess` or an `EventSchedChange` (see figure 5.5).

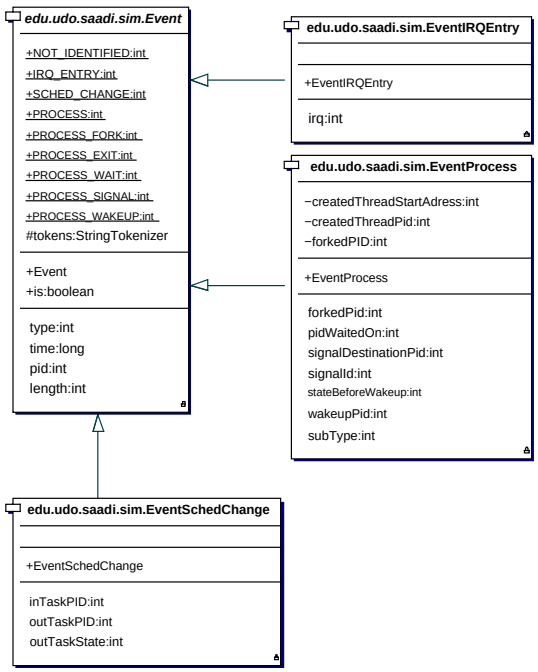


Figure 5.5: UML diagram of the implemented events

These events create a representation for process execution and scheduling. In this regard the class `Statistic` contains a hashtable of `ProcessStatistic` objects, one for each process recognized by the LTT. The `ProcessStatistic` itself contains a list of `Execution` objects, that are created when the corresponding process owned the CPU for a period of time. A class `ResponseTime` represents the time period between an IRQ and a succeeding scheduler change. After having finished analysing events, we compute scheduling criteria for scheduler hierarchy optimization. Since this computation has to be looked at from a more theoretical point of view, we refer to chapter to learn more about that. Intending to enforce hierarchy optimization the data recorded during analysis (and transferred to the `Statistic`) has to be accessible by the ASMGenerator for its next run. For that reason `SAADISim` stores the information of the `Statistic` in the underlying hierarchy.

5.1.3 Parser

by Janni

The `Parser` package builds the foundation for this project. It is used in several locations in the

project. The main purpose of this package is to read and parse a SDL file and create the corresponding Parser objects for every SDL-tag. These objects are then used by the `ASMGenerator`, `GUI` and `CodeGen` modules. The Parser is implemented using the JAXP framework of SUN JDK 1.4.x.

The main method of the `Parser` package is `parseFile(String xmlFile)`. This method opens the SDL file `xmlFile` and parses its contents. For every tag in the SDL file an object is created, these objects/classes correspond to the SDL elements. All attributes of a SDL tag are stored inside the corresponding `Parser` object. The `Parser` checks for invalid data types and also performs range checking for every attribute to enforce syntactic analysis. When the `Parser` detects errors it displays a detailed error message describing the line in the SDL file where the problem appeared, and which tag/attribute was responsible for the problem. The attributes in a `Parser` object can be modified by using the get/set-methods of the specified class. Example: You want to work with the `CacheSize` attribute of the `Processor` class:

```
Processor.getCacheSize( );
Processor.setCacheSize( 128 );
```

Here a little list with the `Parser` subclasses and their SDL equivalents:

- `HardwareConstraints.java` - This class corresponds to the <HARDWARE_REQUIREMENTS> tag in SDL and contains a list of `Architecture` objects.
- `Architecture.java` - This class corresponds to the <ARCHITECTURE> tag in SDL and also contains a list of `Processor` objects
- `Processor.java` - This class corresponds to the <PROCESSOR> tag in SDL
- `SoftwareConstraints.java` - This class corresponds to the <APPLICATION_REQUIREMENTS> tag in SDL and has a list of `Application` objects
- `Application.java` - This class corresponds to the <APPLICATION> tag in SDL and also contains a list of `Thread` objects
- Threads:
  - There are three types of threads a user can specify in the SDL: `ThreadBatch`, `ThreadInteractive` and `ThreadMultimedia`.
  - Example for a <INTERACTIVE> tag which corresponds to the `ThreadInteractive.java` class:

    ```
    <INTERACTIVE id="3" importance="1" response_time="10"\\
    response_time_unit="ms"/>
    ```

  - Example for a <PERIODIC> tag which corresponds to the `ThreadPeriodic.java` class:

    ```
    <PERIODIC id="6" importance="3" amount="50" \\
    amount_unit="ms" period="700"/>
    ```

  - Example for a <BATCH> tag which corresponds to the `ThreadBatch.java` class:

```
<BATCH id="2" importance="2" turnaround_time="110"\\
 turnaround_time_unit="ms" share="50"/>
```

The `Parser` package object additionally contains code to write a `Parser` object hierarchy back to disk. This is mainly used by the GUI to save a modified SDL file back to disk. By calling `Parser.saveToStringBuffer( )` you get a `StringBuffer` object which contains the SDL file.

Another variation is the method `Parser.saveToHTMLBuffer( )`. This method saves the `Parser` object hierarchy in HTML format using syntax coloring for the different elements. As a result you get a `StringBuffer` containing the generated HTML code.

The SDL_ID is generated out of the normalized SDL file text. The normalization is a process where all white spaces are deleted and all text is converted to lowercase. The order of all attributes and tags is fixed. This normalized SDL is then used to generate an MD5 key which is used as the ID of the current SDL file.

At last, a very important thing about the `Parser` object is that the SDL DTD is not read out of a file. The DTD is stored inside the `Parser` object to prevent manipulations of the DTD. Some users could try to manipulate the DTD and that would cause the Parser to crash or to throw error messages. The DTD is stored as a `StringBuffer` object inside the `Parser` class.

5.1.4 Saadi Database

The function of the SAADI Database is to save the SDL file, the generated and tested hierarchy including statistics about its fitness-values. Basically, the SAADI Database is merely a set of MySQL database tables.

It consitst of two main components:

- the DB itself,
- the SAADI interface (API), that allows for access to the SAADI db

The Database structure:
The Database consists of two tables.

Id(String)	XML(String)	averageFitness(double)

Table 5.1: SDL TABLE

In the first table called sdl, the sdl files will be saved as strings(varchar 255), which can be accessed according to their sdl id, and the fitness value average will be saved as double.The average of the fitness values of all hierarchies is the arithmetic average of number of hierarchies and sum of the fitness values.

The second table named hierarchy consists of four columns (id_sdl varchar[255] , id_hierarchy int[50], hierarchy of type longblob, fitness of type double). In the first column the sdl_ids are saved. The second column is dedicated to the id_hierarchy which respects the order of generation of the

Id_sdl(String)	Id_hierarchy(int)	Hierarchy(longblob)	Fitness(double)

Table 5.2: Hierarchy TABLE

hierarchies. The third column is dedicated to the hierarchy itself which is saved as type longblob and which can be accessed by the relationtuple (id_sdl, id_hierarchy). The fourth column represents the fitness value of the hierarchy saved in the same row.

Now we want to give an overview about:

- The Database Interface
- The ConnectionManger and SDLObjectStore class

5.1.5 The Database Interface

As one can see in the uml diagram above, the class ConnectionManager is used by two classes, namely SDLObjectStore und HierarchyObjectStore. The SDLObjectStore class can hold multiple objects of type HierarchyObjectStore. While the ConnectionManager class is the one, that takes care of all the connection handling mechanisms (to the DB), the SDLObjectStore saves the SDL files and works as interface between the SAADI user and the SAADI DB. The function of the HierarchyObjectStore class is to save the hierarchy of type Hierarchy in the DB.

5.1.6 The ConnectionManager and SDLObjectStore class

The SDLObjectStore class uses the ConnectionManager class, which in turn makes uses of the Java Database Connectivity (JDBC) interface to establish DB connections. (public java.sql.Connection getConnection() , public void destroy, public boolean establishConnection() are the maintenance functions).

- Principles of JDBC

  JDBC is part of the Java Standard API. It provides interfaces for access to SQL databases by providing simple interfaces. With the help of JDBC, database clients in applets, servlets, JSP or EJBs can access relational databases. JDBC makes use of object oriented design patterns, but is low level in its function of accessing the database, since one will have to write plain SQL statements in order to make queries. Conceptionally, JDBC is close to OBDC, however JDBC is type-stricter and has a cleaner interface. JDBC helps on platform independent development and provides flexibility for developing database access functionality.

- Design of the JDBC-API

  While most database have specific APIs and features, JDBC tries to generalize this functionlity by providing one generic interface to all database implmementations. This is done by providing a mechanism for loading a database specific driver which encapsulates database specific details like connection handling, authentication etc. If one wishes to access a different database one only needs to load the corresponding driver!

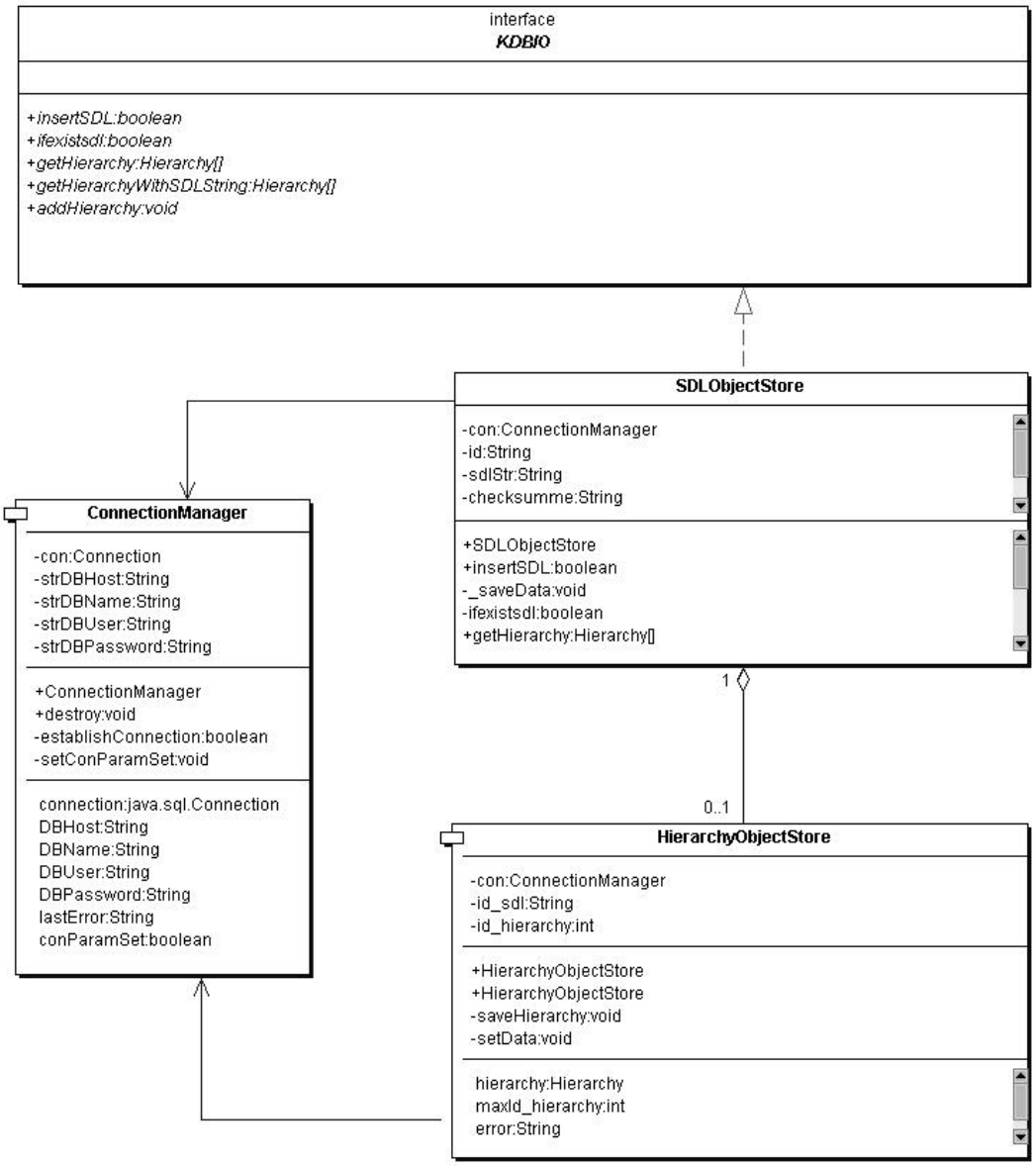


Figure 5.6: UML Diagram of the GUI

- Functionality of the JDBC core

  The functionality of the JDBC core is roughly outlined in the following:

  1. Establish connection to the DB
  2. send SQL-Statements and retrieve the reults
  3. transactions
  4. Stored procedures (these are sql statements similar to functions stored in the database)
  5. Prepared statements
  6. Batch-Execution of multiple sql statements
  7. scrollable results
  8. Functionality to iterate the result set.

- JDBC driver types

  When using JDBC one has to select from four JDBC drivers:

  1. ODBC-Bridge:
     JDBC calls will be compiled into ODBC calls (e.g embedded c-code) Note that both the driver and the native db libraries have to be provided to the client.
  2. Partial Java-driver:
     Compilation of JDBC calls into native db api calls while making use of vendor-specific db apis. Here, only native methods are necessary for the client.
  3. Net Protocol ALL- Java driver:
     JDBC driver, completly implemented in JAVA code- will be downloaded by the client given that specific middleware is provided.
  4. Native Protocol All-Java driver:
     direct communication with the DB server. No native code or ODBC drivers are necessary here. This option is the most modern one.

- Driver interfaces in JDBC

  1. java.sql.DriverManager:
     Registers the driver and establishes connections to the database. This part contains drivers loaded at run-time.
  2. java.sql.Connection:
     represents a database connection.
  3. java.sql.statement:
     Allows for deployment of sql statements.
  4. Java.sql.ResultSet:
     maintains the results of an sql-statement in form of a relation. With this class one can also access every single comlumn of a result.

Usage of the above mentioned classes:

- Driver registration:
  A jdbc driver will be registered by loading the respective driver. /(for MYSQL: mysql-connector-java-2.0.14-bin.jar) Class.forName(,Com.mysql.jdbc.Driver);

- DB-Get Connection:
  database will be reached through URL

```
String  url= ,jdbc:mysql://hostname";\\
String dbName="saadi" String usr="user";\\
Sring   pwd= "passwd";

ConnectionManager con;

Con.setDBHost(url); Con.setDBName(dBName);


Con.setDBUser(usr); Con.setDBPassword(pwd);
Daten aus der DB auslesen:Create an instance of Statement.\\
Statement St= db.createStatement();

//Create an instance of  ResultSet
ResultSet rs=st.executeQuery(,SELECT * FROM hierarchy WHERE
Id_sdl= MD5 Schlüssel ")\\

traverse and read ResultSet  -\\
Demonstration using the getFitnessValues method\\

public double[] getFitnessValues(String sdlId)\\
    {
        double[] flist = new double[selectCount(sdlId)];
        try
        {
            Statement st = \\
            con.getConnection().createStatement();\\
            ResultSet rs;\\
            PreparedStatement pst =
                con.getConnection().prepareStatement(
"select fitness from hierarchy where   " + " id_sdl = ?");
            pst.setString(1, sdlId);
            rs = pst.executeQuery();
            for (int i = 0; rs.next(); i++)
            {
                flist[i] = rs.getDouble("fitness");
            }
            pst.close();
        }
        catch (SQLException se)
```

```
        {
        }
        return flist;
    }
```

It gets a SDLID and returns a field of FitnessValues of all hierarchies,
whereby the run of Resultset happens over:
for (int i=0; rs.next(); i++)

by Ammar, Abdulwhab

5.1.7 Graphical Users Interfaces

by Janni

By reading the other chapters of this report it is easily perceived that SAADI tries to model complex scenarios. Nevertheless we think that it would come in handy if we made our system easily accessible and useable. You can use SAADI in two different ways: either you can choose console mode or alternatively you can use the SAADI-GUI.

When you use the console mode, you have to manually edit the SDL file which is very error prone because you can easily make mistakes in the SDL syntax even when you use a dedicated XML editor. Then you can parse the SDL file and generate an ASM hierarchy and the resulting scheduler code without actually having any idea about the generation process.

The SAADI GUI is able to easily visualize the whole process from creating/editing a SDL file, generating an ASM hierarchy, generating the scheduler code and finally recompiling the kernel. Everything is under one GUI and you have full control of every step in the SAADI generation process.

The GUI is also able to speed up the workflow between creating/modifying a SDL file and generating the resulting ASM hierarchy. The common user will generate/edit a SDL in the GUI, create the ASM hierarchy on the fly and view the generated ASM hierarchy graphically as a tree. Then he can choose the single threads or schedulers in the ASM view and let the ASM frame display information about every object in the ASM hierarchy. The ASM hierarchy allows the user to select a thread in the ASM frame and modify it in the SDL frame. This way the user can very quickly fine tune the ASM hierarchy by modifying the SDL file in the SDL frame, generating the ASM view, modifying the SDL, regenerate the ASM view and so on.

The ASM frame and the SDL frame are connected with each other: modifications or mouse clicks in one frame affect the other frame, e. g. when you click on a thread in the SDL frame the corresponding thread in the ASM frame will be highlighted and vice versa.

After the user is satisfied with the SDL file he created, he can choose to generate the Scheduler code for the current SDL file, then the GUI automatically creates all necessary intermediate files and objects. In the future, the GUI will also be able to bind the scheduler code into the linux kernel and

recompile the kernel.

Error messages from the parser, asm and codegeneration modules are caught by the GUI and presented in dialogs thus giving the user full visual feedback of every mistake that could occur.

5.1.8 Main Components

Let's have a look on how the SAADI-GUI is structured and what components are used for it: The SAADI-GUI's structure is quite simple, it consists of three main components.

- the MainFrame is the main window and operates as a container for all other frames which are MDI children and are opened inside the MainFrame. It is also responsible for handling the main menu actions and for all speed button events.
- the SDLFrame is responsible for displaying and modifying the current SDL file in a tree form.
- the ASMFrame displays the ASM hierarchy that is generated from the current SDL file. It is able to display the hierarchy in different ways and has some other features that will be explained in detail later in this section.

5.1.9 MainFrame

The MainFrame is the main window of the SAADI GUI. It is generated, displayed and centered on the screen by the main class of the GUI, GUI_Main.

MainFrame is responsible for generating the toolbar with the speed buttons and for the main menu entries. This class handles all main menu events and all speed button events. It also takes care of enabling/disabling menu items and speed buttons in different situations. For example when you start the GUI and no SDL file is opened, the MainFrame class will disable all menu items and speed buttons for saving the SDL, generating the ASM hierarchy, deleting an SDL item etc. It also handles all graphical icons used in the GUI.

It is also responsible for handling all child windows like the SDLFrame, ASMFrame, Codegen-Frame, SourceCodeFrame etc. Those frames are displayed inside the MainFrame and one cannot move them out of the MainFrame. MainFrame provides methods for cascading, tiling and switching of child frames.

5.1.10 SDLFrame

In general the SDLFrame allows the user to create a new SDL file or to modify an existing SDL file. It also provides the possibility to create a random SDL file which is generated by a specialized algorithm and is used for testing purposes.

The SDLFrame is split horizontally in two panels:

- The left panel consists of a JTree component in which all elements of the SDL file are represented by a tree node. The tree mimics the hierarchy of the SDL-file. Different nodes represent the hardware requirements, architecture, processor, software requirements, application, psthread, resthread and tsthread elements of the SDL.
- the right panel shows the details of every node in the JTree component. When you select a processor in the JTree you can view and modify all of it's attributes in this right panel. All attributes are editable by text fields and by spin edits. The GUI only allows valid values for the attributes. This way the user cannot enter invalid values that would lead to errors in the ASM hierarchy generation process. After the modifications are finished, one can click on the "Apply Changes" button to let the GUI save the modified SDL element back into the internal parser object. At this stage the GUI is internally checking if the modified SDL object is valid or if there are any collisions with other SDL elements or with other attributes in the same SDL object. When such a collision occurs, the GUI presents an error message with a detailled description of what went wrong and suggestions on how to fix the problem.

All SDL elements in the JTree have a leading icon which visually shows the differencs between these objects. The thread objects icons are coloured boxes, every thread type has it's own color to let the user easily distinguish the thread type.

- RESThreads are displayed using the color cyan
- PSThreads are displayed in yellow
- TSThreads are green

These colors are customizable and can be changed in the options menu. This color scheme is used throughout the whole GUI in order to achieve a consistent color layout that makes all threads easily recognizable. The user will find this color layout in the SDLFrame, the ASMFrame and in all dialogs that let you manipulate thread objects.

The user can only add/delete application or thread objects. To do so one can use the two speed buttons on the toolbar. When you add a thread the add dialog let's the user choose which type of thread he wants to create, again using coloured icons for the different thread types. All newly created threads and applications have safe default values.

The same goes for new SDL files: When one chooses to create a new SDL file via the F̈ile - New SDL file m̈enu or via the corresponding speed button, the MainFrame generates a SDL file which only consists of a hardware requirements object, an architecture object with default values for the INTEL architecture and with a processor of the INTEL processor family. This newly generated SDL is a valid SDL and can be compiled and used without further modifications.

5.1.11 ASMFrame

The ASMFrame consists of a graphical area in which the ASM hierarchy of the current SDL file is displayed and of an area with different check boxes and buttons.

This frame generates an ASM hierarchy from the current SDL file and provides different ways of visualizing it.

The generated ASM hierarchy is displayed in form of a tree with coloured nodes and connection lines. Elliptical nodes represent scheduler objects and rectangular nodes represent thread objects. Again, the same color scheme like in the SDLFrame is used for the different objects.

By clicking on a thread object the user can highlight the corresponding SDL object in the SDLFrame. This connection works in both directions. The user can also choose an Application in the listbox at the bottom of the ASMFrame and the ASMFrame then highlights all threads that belong to the selected application. This also works in the SDLFrame.

The ASMFrame not only lets the user view the generated ASM hierarchy, but also shows the incoming guarantees for every ASM object. The user can see the guarantees for every ASM object by moving the mouse cursor over an ASM object. A tooltip text will appear showing all relevant informations about this object.

The ASMFrame is able to show the ASM hierarchy in two different views:

- vertical view: in this view the ASM hierarchy is shown vertically. This means that all children of a scheduler are displayed vertically above or on top of this scheduler.

- horizontal view: the children of a scheduler are displayed horizontally to the left or the right of this scheduler.

The ASMFrame reflects the applied modifications in terms of a scheduling hierarchy. The user can also change the views look and feel by adjusting color settings or by choosing between horizontal and vertical views. In addition to that, when moving over a thread with the mouse the user will be shown the guarantees this thread will receive. Finally while clicking an application or thread the corresponding items will be highlighted within the view. As a special gimmick the user can save the displayed scheduling hierarchy in JPEG format for further external usage.

The last option in the ASMFrame is the ability to export the current ASM hierarchy into a JPEG file. The user clicks on the S̈ave-button in the lower right area of the ASMFrame, a dialog pops up, the user enters the desired JPEG reolution and the filename and the ASMFrame generates a JPEG with the specified dimensions for further investigations.

5.1.12 SDLSourceCodeViewer

The user can view the SDL-File source code and the corresponding HTML code within this viewer, though the user cannot make any changes to the source code - the SDLFrame is meant to do this. Once again heavy usage of color is being made here.

Sometimes it is necessary to view the XML code of the current SDL file. The GUI offers a special dialog that shows the XML code of the current SDL file, allowing the user to stay in the GUI instead of switching to an external application to view the code.

The SDL code is displayed with syntax coloring, that means that the different parts in the code are coloured differently to make the code more apprehensible.

The user can change this color scheme in the options dialog of the GUI.

5.1.13 CodeGen Source Code

Here the generated code from the CodeGeneration phase will be displayed. This C-code is the result of the processing of an ASM hierarchy, which was generated from a SDL-File.

The SAADI GUI can also generate source code from the current SDL file.

To do so the user clicks on B̈uild - Generate code" or onto the corresponding speed button on the toolbar.

The GUI now checks the validity of the current SDL and in case of errors a detailled description is displayed with suggestions on how to fix the problem. Then the ASM hierarchy is generated. When this happened correctly the GUI starts the CodeGeneration. In this phase the priorly generated ASM hierarchy is translated into C - code. The generated C-code is displayed in the CodeGenFrame allowing the user to have a look at the generated code and if necessary make modifications in the SDL file. This workflow makes the whole process from generating a SDL file, then the ASM hierarchy and then the C-code less error-prone, easier and faster.

5.2 The Code Generation-Class

by Christian Bockermann

As described later, this class generates the code of setup function that initializes the scheduler hierarchy which is called at late kernel initialization. In addition to that it generates code for a function that returns the appopriate scheduler for a task and thus implements the application mapping.
This sections does not deal with the initialization functions but describes the internals of the CodeGeneration class and the methods exported to the rest of the system.

5.2.1 Template-file

The CodeGenerator makes use of a template of `saadi_init.c` which is distributed in the Java archive file `SAADI.jar`. After loading this file using the classloader the essential parts of the two functions are inserted. These parts contain the application mapping and the initialization that consists of several `saadi_construct_scheduler` calls.
Basically this template file contains certain marks (e.g. SAADI_APPLICATION_MAPPING) which are later replaced by the content of specific string buffers (private buffers of the CodeGenerator-class).

5.2.2 Public methods

In order to generate code, the CodeGenerator offers a few public methods which are exported to the other classes of the SAADI-project. These are the entry-points where the classes start to use this CodeGenerator.
Before generating any code an instance of the CodeGenerator-class has to be created. Instatiation is done by the use of a public constructor which requires two different parameters: The ASM-hierarchy created by the ASM-Generator and a status of the ASM-Generator. The process of code generation is started by a call to `generateCode()`-method of the instantiated object. This method recursively traverses the hierarchy and fills the internal string buffers of the object with code for each task and scheduler found in the hierarchy.
After the traversal the different string buffers contain the C-calls neccessary to construct the scheduling hierarchy and the application-mapping code (this the part cannot be hard-coded). Next the marks

in the template-file are replaced with the contents of the string buffers and the result is print out to standard output. In parallel it is saved to `kernel/saadi_init.c` relative to the `$LINUX_ROOT` specified at the command line.

5.2.3 Validation

A special feature of the CodeGenerator class is that it checks a directory containing the linux source code for a valid SAADI-enhanced kernel. This is done by looking for specific SAADI- related files which depend on the generated ASM tree. The method `checkSourceTree()` returns true if all scheduling algorithms used in the ASM tree exist in the linux source (to be precise: the files for the scheduling algorithms have to exist).

5.2.4 Implementation

The implementation of the CodeGenerator is not very complex. It basically consists of a tree-traversal and the decision between scheduler and task. Figure 5.7 shows a pseudo-code notation of this traversal.

```
generateCode (root,schedulable) :

if schedulable is Scheduler then
   addScheduler(root, schedulable)
   k = schedulable.getNumberOfChildren()
   for i = 0 to k do
      generateCode(schedulable, schedulable.getChild(i))
   end for
else
   addTask(root,schedulable)
end if
```

Figure 5.7: Pseudo-code for the code-generation

5.3 Kernel

The changes to the linux kernel to integrate our hierarchical scheduling framework are mostly contained to the ”kernel/sched.c“ file which is the place where all of the scheduling logic is implemented. All central data structures are defined in the ”include/linux/saadi.h” file. Some other files were also modified to support our changes but these modifications are minor and will be pointed out at appropriate places in this text. Our integration work is based on the 2.5.58 development series kernel.

5.3.1 Framework

by Piotr

The "schedulable" structure

Our whole hierarchical scheduling framework is centered around the `schedulable` structure so the documentation of our kernel work shall start here. The `schedulable` structure encapsulates all data the framework needs to schedule an object - either a task or a scheduler. Tasks are represented by the structure `task_struct` (defined in "include/linux/sched.h") in the standard linux kernel. In order to keep the changes to this critical data structure to a minimum (to avoid conflicts in the future) we only added one further attribute - `this_schedulable` which allows us to refer to the underlying `schedulable` structure. Besides this change only some scheduler contained, now unneeded attributes were removed.

Listing 5.1: The schedulable structure

```
struct schedulable {

        scheduler_t *parent;

        long int time_slice;
        int prio;

        /* status of the schedulable */
        unsigned int status;

        union {
                struct linux_data       O1;
                struct eevdf_data       eevdf;
                struct res_data         res;
                struct resps_data       resps;
                struct ps_data          ps;
        } sched_data;

        scheduler_t     *sched;
        task_t          *task;
};
```

Every `schedulable` object has a `parent` scheduler which is the scheduler used for scheduling this object (note that a schedulable object can be associated to both tasks and schedulers; they are regarded as the same from the scheduling point of view). The `time_slice` attribute denotes the cpu time in ticks left to the object and is decremented via the timer interrupt (which depends on the definition of the `HZ` value which can change from architecture to architecture and is set to milliseconds on i386). If it reaches zero then a rescheduling is initiated. The next attribute - `prio` - is a compatibility hack to allow linux system calls (e.g. `sys_nice()`) to access to the prioritity of the task/scheduler. If the object is scheduled by any other scheduler than the saadi-port of the original linux scheduler than changes in this value will have no effect. At the moment two different states are defined for a schedulable object:

- ENQUEUED - meaning the schedulable object is currently enqueued in the runqueue of the parent scheduler and

- FIRST_TIME_SLICE - indicating that this is the first time slice the object uses (e.g. a freshly forked process) (this information is useful for providing more fairness to processes which spawn a lot of children that exit very fast)

The `sched_data` union provides scheduler specific information and is unused in the framework itself. If this schedulable structure refers to a scheduler then the attribute `sched` points to an appropriate structure, and is zero if the object is a common task and vice versa for task objects.

The "scheduler" structure

Schedulers are represented by the `scheduler` data structure. Besides encapsulating the obvious attributes `id`, `type` and `class` (denoting the id, type and the class of the scheduler) it provides the framework-scheduler interface by which the scheduler specific logic can be accessed and against which new schedulers can be written and integrated into the framework. The details of this interface will be discussed in the next section.

Listing 5.2: The scheduler structure

```
struct scheduler {

        unsigned int id;
        unsigned int type;
        unsigned int class;

        spinlock_t lock;

        link_func_t                 link;
        unlink_func_t               unlink;
        join_func_t                 join;
        leave_func_t                leave;
        schedule_func_t             schedule;

        runqueue_t rq;

        schedulable_t * this_schedulable;

        /* list of idle child schedulers */
        struct list_head idle_list;

        /* scheduler status */
        unsigned int status;
};
```

The spinlock `lock` guards the `scheduler` structure and the associated runqueue `rq` and serializes the access to it. As with the `task_struct` the `this_schedulable` pointer provides us with the access to the underlying `schedulable` data structure. In order to avoid wastage of precious cpu-cycles for large scheduling hierarchies, our framework automatically removes idle schedulers (i.e. schedulers which have no tasks in their runqueue and only idle schedulers beneath them) from

the hiearchy and temporarily places them on the idle-list of their parent scheduler. Obviously the root scheduler of the hierarchy can not become idle and the idle-task which is a special system thread not handled by any scheduler is scheduled on the cpu if there are no other tasks to schedule. A scheduler is reinserted into the hierarchy when a task associated with this scheduler is created or wakes up from sleep. The `status` field helps to keep track of the current status of the scheduler. At the moment only one status is defined:

- SCHED_IDLE - the scheduler has been temporarily removed from the hierarchy and placed on the idle-list of it's parent scheduler

The "runqueue" structure

The `runqueue` data structure is used to keep track of the schedulers and tasks that currently compete for the cpu time based on the rules defined by the scheduler to which the runqueue is attached. The runqueue structure can be devided into two parts: the first part implements attributes that help keep track of different statistics (`nr_running`, `nr_switches`, `nr_uninterruptible`)that are needed by other parts of the system (`migration_thread`, `migration_queue` - these are needed by cpu-migration-code to be found in the "kernel/sched.c" file but not directly connected to our scheduling work) or are used directly by our scheduling framework (`curr` - points to the schedulable currently being run, `nr_idle`, `idle_queue` - both are used by the scheduler (un)-idle code). The second part is scheduler specific and implements the data structures that are needed to hold all the schedulable objects. For instance the port of the original linux scheduler uses two arrays of lists while the eevdf scheduler uses binary trees.

Listing 5.3: The runqueue structure

```
struct runqueue {

        unsigned long nr_running, nr_switches, nr_uninterruptible;
        schedulable_t *curr;

        task_t *migration_thread;
        struct list_head migration_queue;

        /* idle schedulers are removed from the active \\
         / expired queues
         * and put here */
        unsigned long nr_idle;
        struct list_head idle_queue;

        atomic_t nr_iowait;

        /* scheduler implementaion specific data
         */
        union {
                struct O1_rq_data       O1;
                struct eevdf_rq_data    eevdf;
```

```
                struct resps_rq_data resps;
        } impl;
};
```

The “fork()” code path

To illustrate the sematics of the (most important) saadi related functions and provide an high level overview of the inner workings of our scheduling framework we will now have a look at the different code paths taken during the execution of a task.

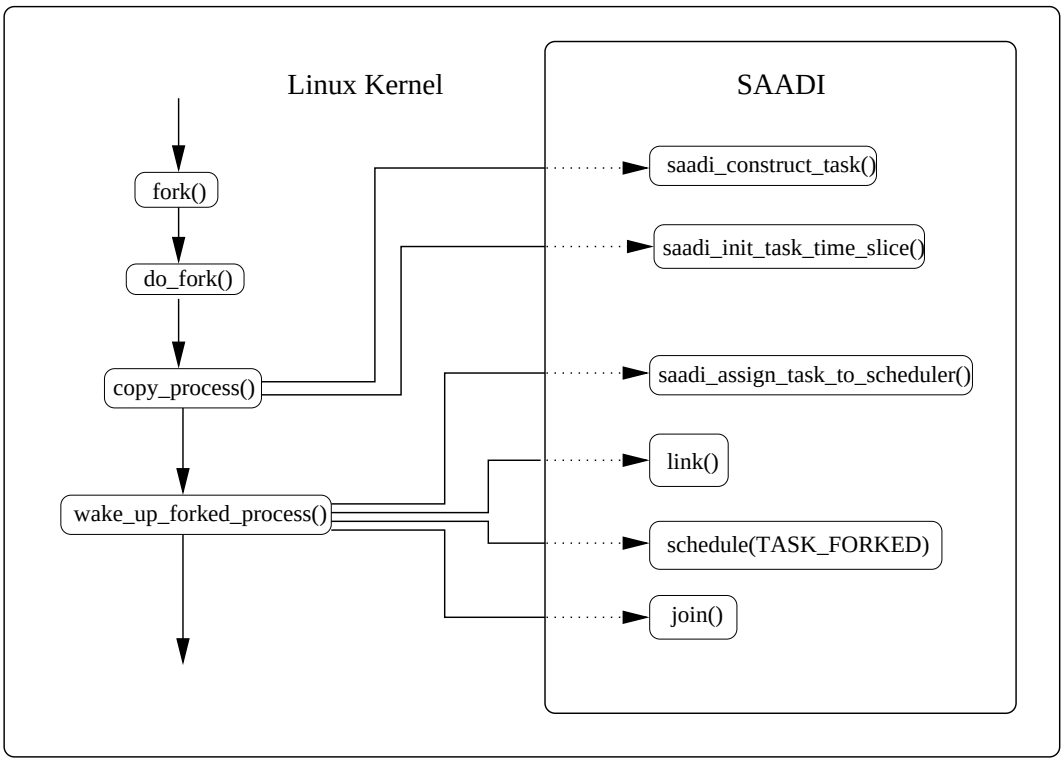


Figure 5.8: The fork() code path

The life of (nearly) every task begins with a call to `fork()` by its parent. During this call various data structures of the parent are copied and new data structures are initialized. The details of this process can be found in “kernel/fork.c:copy_process()”. From the perspective of our framework it is also the right time to initialize the data structures our framework needs to cope with the newly created task - mainly the `schedulable`-structure. This is accomplished through a call to `saadi_construct_task()` late in `kernel/fork.c:copy_process()`. If something goes wrong `saadi_destruct_task()` is called to clean up. We also have to give the task some initial time slice. This is done by `saadi_init_task_time_slice()` which just divides the parents time slice by two and gives one part to the parent and the second to its child. Obviously this keeps the sum of all time slices currently given to processes constant and prevents the parent from artificially extending its usable time slice resulting in more scheduling fairness. When the child process exits very fast without using up its first time slice, the remainder is reclaimed for the parent.

The next step is to wake up the newly created process (i.e. let it join the set of processes currently competing for cpu time) which is implemented by a call to `wake_up_forked_process()` from `kernel/fork.c:do_fork()`. Here our first goal is to determine by which scheduler this task has to be scheduled in order to know into which runqueue to put it. The function `saadi_assign_task_to_scheduler()` implements this by using the name of the task to derive its scheduling parameters as defined in the ASM (note that this function is created by the code generation module). After knowing the responsible scheduler we call its `link()` function to make this connection permanent and give it a chance to react to this event by calling `schedule()` with the `TASK_FORKED` flag set. Lastly we call `join()` to place the task (or rather the underlying schedulable object) into the schedulers runqueue. Now the task can be scheduled like any other task currently running on the system.

The “exec()” code path

Since running a lot of copies of the same task is not really that interesting hence a task has also the opportunity to replace the code/data segments inherited from its parent with newly loaded data from the hard disk (e.g. your bash does this when you execute a new program). The responsible syscall is called `exec()`. Due to the fact that in our system a task is identified by its filename a call to exec implies that the connection between the task and its scheduler has to be re-evaluated. This is accomplished in `fs/exec.c:do_execve()` by a call to `saadi_task_reassign()`. If we have to move the task to a different scheduler we first remove it from the runqueue of its old scheduler (`leave()`), then cut the connection to this scheduler (`unlink()`), create a connection to its new scheduler (`link()`) and finally end the movement by placing it in the new scheduler’s runqueue (`join()`).

The “scheduling” code path

The scheduling process is normally initiated from `sched.c:scheduler_tick()` which is called from the timer interrupt with `HZ` frequency and gives us our basic cpu time metric - the tick. Of course a request for a new scheduling decision can also be initiated by other parts of the system (e.g. when a task is forked). To signal the need for a new scheduling decision, a call to `set_tsk_need_resched()` is sufficient. In `sched.c:scheduler_tick()` we decrement the time slice left for the currently running task and also every scheduler on the path from its parent to the root scheduler of the hierarchy by one. If any time slice reaches zero we signal that a new scheduling decision is needed. The linux kernel responds to this need by calling `schedule()` as soon as possible. Here a rather large loop is used to traverse the scheduling hierarchy from the first scheduler where a new scheduling decision has to be made on the way down from the root scheduler until we finally arrive at a leaf of the hierarchy which is always a task. Note that only tasks are objects that are given the cpu by a context switch. Scheduler code is only executed in the context of our framework, in order to determine a task that can be finally scheduled and maintance of our data structures. The whole process is made complex by the fact that scheduler runqueues can be empty in which case we put the scheduler on the idle-list of its parent scheduler and in the case of the root scheduler we have no real task that we can schedule and therefore we schedule the idle-task.

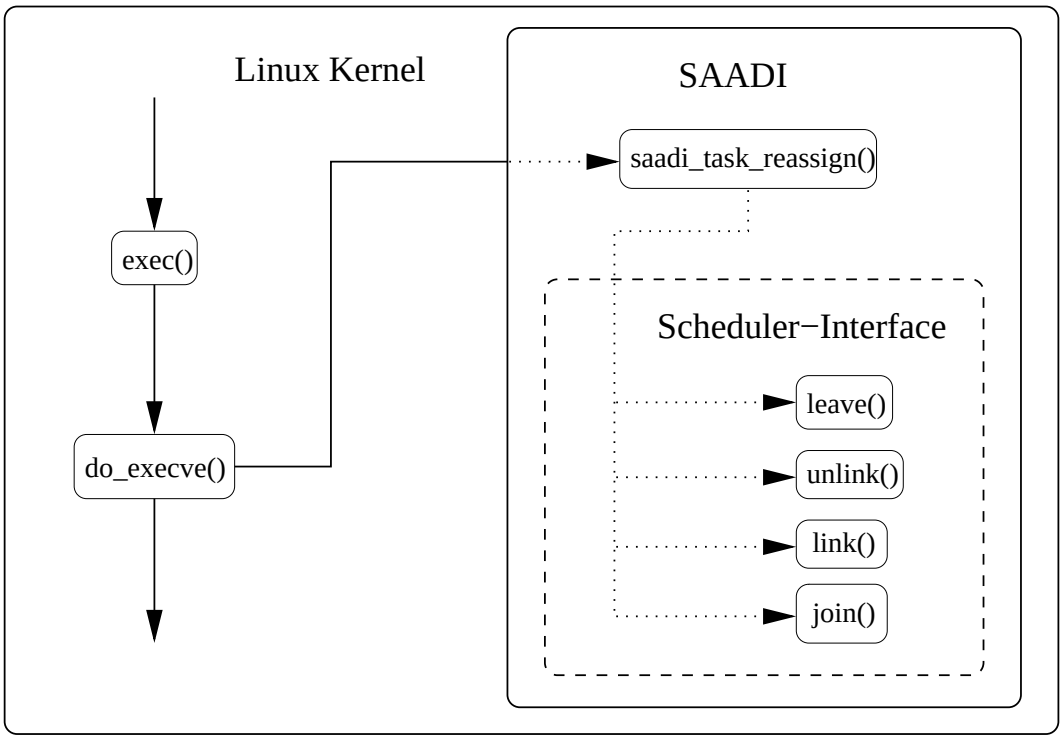


Figure 5.9: The exec() code path

5.3.2 Interface

by Piotr

- link()
  It is called whenever an association between a scheduled object and its scheduler changes. This happens for example when a freshly forked task is assigned to a scheduler or a task (or a scheduler) is moved in the hierarchy (then we have to call ”unlink()” first on the old scheduler). The specific scheduler can use the call to initialize the scheduling specific ”sched_data” union in the assigned schedulable object or setup some intern data structures etc.

- unlink()
  As can be expected unlink is the counterpart to link and called when the link between a schedulable object and its parent scheduler has to be removed (e.g. when the task exits or when a schedulable object has to be moved in the scheduling-hierarchy)

- join()
  join() mostly replaces the ”activate_task()” function in the standard linux kernel. It is called when a schedulable object joins the runqueue of the scheduler (note: ”link()” must have been called prior to that). After joining the runqueue a schedulable object competes with the other objects in the runqueue of the scheduler for the cpu and if it wins it is given the cpu for a period determined by the scheduler. Note that ”join()” and ”leave()” are called more frequently as one

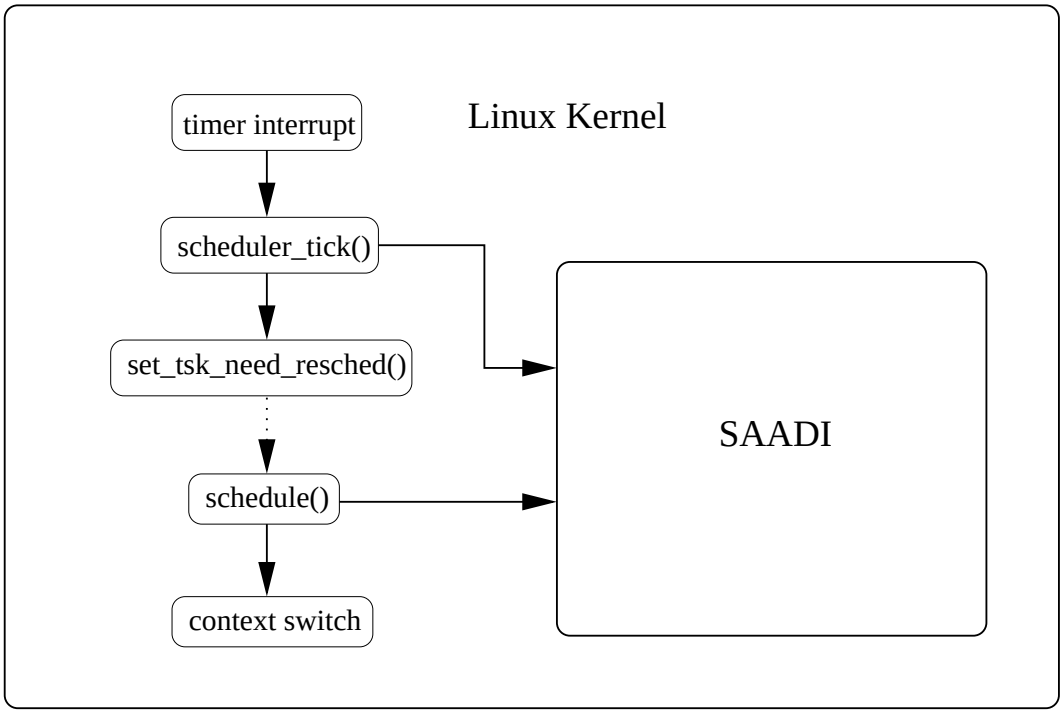


Figure 5.10: The schedule code path

would expect. This is due to the fact that on every waiting of the task in the system (e.g. when waiting for an IO-operation to complete) the task (or the underlying schedulable object to be precise) is removed from the runqueue of the parent scheduler and is placed in the appropriate wait queue.

- leave()
  This is the counterpart to ”join()”: a schedulable object leaves the runqueue of the parent scheduler. It is important to understand that while ”unlink()” is mostly called when a task exits, ”leave()” is frequently called while e.g. waiting on some ressource.

- schedule()
  This is the place where the scheduling algoritm is implemented. A special event variable is used to describe the exact scheduling event type that must be answered. The following values and semantics are defined:

  – SCHEDULABLE_EXPIRED
    Indicates that the time slice of the schedulable object has reached zero.
  – SCHEDULABLE_PREEMPTED
    Denotes that the schedulable was interrupted on a sub time-quantum level. This can be useful in realtime enviroments where one does not want to wait until the time quantum of the schedulable object is used up or it yields the cpu. This event is not used at the moment.
  – DISPATCH
    Signals the need to choose a new task object for assigning to the cpu. If the returned

object is of type task, then it can immediately run on the cpu. If it is a scheduler, then the framework will recursively issue a DISPATCH event until a task is found. Failure to find an appropriate task results in the scheduling of the idle-task.

– TASK_FORKED When a newly forked task is woken up in ”wake_up_forked_process()” this event is used.

5.3.3 Scheduler Implementation

by Rolf

Currently there are six schedulers implemented in SAADI: roundrobin, fixed priority, linux, proportional share, EEVDF, reservation scheduler and Reservation Probabilistic Soft (RESPS).

The Linux Scheduler

The Linux scheduler uses the same algorithm as the O(1) scheduler in the Linux 2.5 kernel, developed by Ingo Mjolnar. The runqueue data shown in 5.5 consists of two arrays: active and expired, that hold the tasks, sorted by priority. A new tasks is assigned a timeslice and queued in the active array. When the task has used its timeslice, it is assigned a new timeslice and enqueued in the expired array. Once the active array is empty the active and expired array are swapped.

Listing 5.4: per schedulable data for Linux scheduler

```
struct linux_data {

        int prio;
        prio_array_t *array;

        struct list_head run_list;

        unsigned int first_time_slice;
};
```

Listing 5.5: runqueue data for Linux scheduler

```
struct O1_rq_data {

        unsigned long expired_timestamp;

        prio_array_t *active, *expired, arrays[2];
        int prev_nr_running[NR_CPUS];
};
```

PS

The per task data for the PS scheduler is in the “struct ps_data” shown in figure. The share is the Guarantee parameter settable from the user space by a syscall or the scontrol program. The virtual

time vt for a task advances proportional to the time the task is executed and inverse proportional to the share of this task. If a task has been running for "time_running" the virtual time is set to

$$VT = VT + \frac{time\_running}{share} \tag{5.1}$$

The virtual time of a tasks that have been blocked is set to the minimum virtual time of the tasks in the runqueue, so a task can not accumulate credits by sleeping or waiting for IO. The scheduler always chooses the task with the smallest VT to run next. So every task gets execution time proportional to its share. Figure 5.7 shows the PS scheduling algorithm in pseudocode.

Listing 5.6: per schedulable data for PS scheduler

```
struct ps_data {
        int share;
        unsigned long vt;     /* virtual time*/
        unsigned long start_time ;
        struct list_head run_list ;
};
```

Listing 5.7: The PS Algorithm

```
/* schedulable s joins the runqueue*/
ps_join (s){
  if (s.vt>minqueue_VT)
    s.vt = minqueue_VT;
  enqueue(s); /* add s to runqueue*/
}


/* schedulable s leaves runqueue */
ps_leave (s){
  s.vt += runtime/s.share
  dequeue(s); /* delete s from runqueue*/
}


/* s is preempted or timeslice expired*/
ps_preempted(s){
  /* update the VT for s, which has been running for runtime*/
  s.vt += runtime/s.share;
}


/* select new schedulable to run next*/
ps_dispatch (){
  /* select schedulable from runqueue with smallest vt*/
  next = get_request ();
```

```
  next. timeslice = TIMESLICE;
  return next;
}
```

EEVDF

The Earliest Eligible Virtual Deadline First algorithm also provides a proportional share guarantee to its tasks. The share of a task along with other per taks parameters is in the struct eevdf_data shown in figureref:. The per scheduler data in eevdf_rq_data is shown in figureref:. The scheduler has a virtual time, which advances every time a task has run for time delta as

$$V(t) = V(t) + \frac{delta}{total\_weight} \tag{5.2}$$

. Where total_weight is the sum of the shares of all active tasks.

The variables ve and vd denote the virtual eligible time and virtual deadline of a task. When a task makes a request for runtime of requestsize and joins the runqueue its eligible time is calculated as

$$ve_1 = V(t_0) \tag{5.3}$$

the following ve are calculated as

$$ve_{k+1} = ve_k + \frac{request\_size}{w_i} \tag{5.4}$$

. The virtual deadline is

$$vd_k = ve_k + \frac{request\_size}{w_i} \tag{5.5}$$

.

A task will only be scheduled, when its eligible time ve is greater or equal than the queue virtual time V(t). The algorithm select always the task with the earliest virtual deadline vd which is eligible $ve >= V(t)$.

Listing 5.8: eevdf data

```
struct eevdf_data {
        int     share;          /* proportional share of the
                                 * client */

        /* data needed for algorithm implementation */
        int     lag;            /* lag of the client */

        int     ve;             /* virtual eligible time */
        int     vd;             /* virtual deadline */
        int     min_vd;         /* minimum virtual deadline in
                                 * whole subtree
                                 * only used by *_request procs
                                 */
```

```
        schedulable_t * client ;   /* client on which behalf the
                                    * request is made */

        /* pointer for binary tree representation */
        eevdf_data_t * left , * right , * parent ;
};


struct eevdf_rq_data {
        /* the binary tree with all pending requests */
        struct eevdf_data    * request_tree ;

        /* current virtual time */
        int                    virtual_time ;

        /* total weight of all active clients */
        int                    total_weight ;

        /* time in jiffies of last vt update */
        unsigned long int      last_vt_update ;

        /* nr. of ticks we had to forward the virtual time by */
        unsigned long int      nr_vt_forward_ticks ;

        /* nr. of time we had to reset the vt to\\
         prevent overflowing */
        unsigned long int      nr_vt_resets ;
};
```

Listing 5.9: eevdf algorithm

```
/* schedulable s joins the runqueue*/
eevdf_join (s){
  scheduler->total_weight+=s.weight;
  s.ve = scheduler. virtual_time ;
  s.vd = s.ve + ( REQUEST) / s.share
   insert_request (s);
}


/* schedulable s leasves runqueue */
eevdf_leave (s){
  schedule. total_weight -=s.weight;

   delete_request (s);

}
```

```
/* s is preempted or timeslice expired */
eevdf_preempted(s){
    delete_request (s);
   s.ve *= requestsize / s.share;
   s.vd = s.ve + requestsize / s.share;
    insert_request (s);
}


/* select new schedulable to run next */
 eevdf_dispatch (){
   next = get_request ();
}
```

Reservation

The reservation scheduler provides basic soft reservation guarantees. Schedulables scheduled by the reservation scheduler have the guarantee parameters amount and period, held in the “struct res_data” 5.3.3. The scheduler uses an earliest deadline first algorithm similar to the scheduler in the Nemesis OS []. The deadline is always the end of the current period. The scheduler simply runs the task with the earliest deadline for the given amount of time. Tasks which have run for their amount are put in a waiting queue and are reinserted in the runqueue at the end of their period and given a new timeslice as their new amount. If there is no task in the runqueue because all tasks have been executing for their reserved amount, the scheduler runs a random task from the waiting queue for a short period.

```
struct res_data {
        int     period;
        int     exec_time;

        int     deadline;
        struct  list_head   run_list ;
};
```

Listing 5.10: RESBS algorithm

```
/* schedulable s joins the runqueue */
 resbs_join (s){
   s.deadline = now + s.period;
   s. time_slice = s.exec_time;
   enqueue(s); /* insert s into runqueue, sorted by deadline */
}


/* schedulable s leaves runqueue */
 resbs_leave (s){
```

```
  dequeue(s ); /* delete s from runqueue*/
}


/* s is preempted or timeslice expired*/
resbs_preempted(s){
    tasks_in_waiting_queue_with_deadline_over_back_to_runqueue ()
  /* s has used its timeslice insert into waiting queue*/
  dequeue(s,runqueue);
  enqueue(s,waiting_queue );
}


/* select new schedulable to run next*/
 resbs_dispatch (){
  /* return the task with earliest deadline from runqueue*/
  next = get_next (runqueue);
  if (! next ) {
    /* as there is no task left in runqueue, \\
    run task from waiting for short time(overun time)*/
    next = get_next (waiting_queue );
    next . timeslice =5;
  }
}
```

5.3.4 The SAADI System Calls

by Markus Wuebben

In order to provide an interface to the scheduler core within the kernel we implemented a set of system calls to retrieve information about and set parameters within the scheduler[1]. This way a process may set its desired scheduling behaviour and retrieve information about its scheduling parameters and the scheduler available in the SAADI system from userspace.

Why new syscalls?

In fact there are quite a few scheduler specific syscalls available within Linux due to the POSIX 1.b standard and we thought about extending them since they do not suffice the requirements of SAADI but after our research we recogized that the cleanest and ”safest” way to implement the desired functionality would be to implement new calls.

System calls in Linux on x86 architecture

System calls provide the interface between a process in userspace and the operating system (kernel). System calls within Linux on x86 architecture work by putting the the *syscall number* of the desired system call in register *eax* and its arguments into registers *ebx, ecx, edx, esi, edi (,ebp)* depending on the number of arguments the syscall takes [2]. The kernel then handles the software interupt to serve the request. The return value is stored in register *eax*. The necessary assembler code is provided in form of macros which are stored in the `asm/unistd.h`. Listing 5.11 depict such a macro for a one argument system call.

Listing 5.11: A one argument Linux system call macro

```
#define _syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
    : "0" (__NR_##name),"b" ((long)(arg1))); \
 __syscall_return(type, __res); \
}
```

libsaadi and scontrol

In order to provide some convinience for the user of a SAADI enhanced system we implemented a library called *libsaadi*. This library encapsulates the SAADI system calls and features other neat functions. Also, a tool called `scontrol` is delivered with every distribution of SAADI. With the help of `scontrol` the owner of a process (or root) may manipulate the processe's scheduling behaviour. Please refer to the help of scontrol by calling `scontrol --help`. `scontrol` and `libsaadi` can be found beneath the `contrib` directory in your SAADI Linux kernel source tree.

[1] In earlier versions we made use of the so-called procfs available under Linux but this mechanism has been deprecated in favor of the system calls.

[2] This applies only to system calls that take less than five arguments. System calls with more arguments still expect the syscall number to be in *eax*, but the arguments are arranged in memory and the pointer to the first arg is stored in *ebx*.

The New Calls

In Listing 5.12 one can find the prototypes of the implemented system calls. This is how they can be accessed from `libsaadi`. The prototypes can be found in the header file of `libsaadi`. In the Listings 5.13, 5.14, the according parameter definitions can be found.

Listing 5.12: The SAADI system call prototypes

```
long  saadi_signal_deadline_failure  ( pid_t  pid );

long  saadi_set_scheduler  ( pid_t  pid ,  saadi_sched_param_t  *param);

long  saadi_get_scheduler  ( pid_t  pid , saadi_sched_param_t  *param);

long  saadi_get_available_scheduler  ( saadi_scheduler_t  *param);

long  saadi_get_scheduler_of_class  ( int  class ,  types_t  *types );

long  saadi_get_scheduler_classes  ( classes_t  * classes );
```

Listing 5.13: The SAADI saadi_sched_param structure

```
struct  saadi_sched_param {

    int  class  ; //  The class  of  the  scheduler
    int  type ;    //  The type  of  the  scheduler
    int  id ;      //  The Scheduler  ID

    //  The following  union  carries  the  possible
    //  scheduling  parameters a  certain  type  of
    //  scheduler  uses
    union {
        // FIXED_PRIORITY and ROUNDROBIN use the same
        //  struct  like  the  LINUX scheduler
        struct  linux_sched_data          O1;
        struct  eevdf_sched_data          eevdf ;
        struct  res_sched_data            res ;
        struct  resps_sched_data          resps ;
        struct  ps_sched_data               ps ;

    } sched_data ;

};
```

Listing 5.14: The SAADI classes_t structure

```
typedef  int  classes_t [SCHED_MAX_CLASS+1];
```

Listing 5.15: The SAADI types_t structure

```
typedef int types_t [SCHED_MAX_TYPE+1];
```

Listing 5.16: The SAADI scheduler system call structure

```
struct saadi_scheduler {

        int id;
        int class;
        int type;
        int parent;
};

typedef struct saadi_scheduler saadi_scheduler_struct ;
typedef saadi_scheduler_struct saadi_scheduler_t [MAX_SCHEDULER];
```

The semantic of the system calls are:

- `long saadi_signal_deadline_failure(pid_t pid);`
  Processes scheduled by schedulers of class RES can signal a deadline failure to the kernel. The kernel may then react to those failures adjusting its scheduling policy for this process. The argument the syscall takes is

  1. `pid_t pid`: The process id of the process that wishes to signal a deadline failure.

- `long saadi_set_scheduler(pid_t pid, saadi_sched_param_t *param);`
  Processes may set the scheduler it is subject to using this syscall. The arguments this system call takes are

  – `pid_t pid`: The process id of the process wishing to set the scheduler.
  – `saadi_sched_param_t *param`: The scheduling parameter to be filled with the according scheduling parameter the desired scheduler takes.

- `long saadi_get_scheduler(pid_t pid,saadi_sched_param_t *param);`
  Returns the scheduling policy and parameters a process is scheduled under. The arguments the syscall takes are

  – `pid_t pid`: The process id of the process the information is retrieved for.
  – `saadi_sched_param_t *param`: The scheduling paramter struct that is filled with the according information. The memory is to be allocated by the calling process.

- `long saadi_get_available_scheduler(saadi_scheduler_t *param);`
  Returns the available schedulers in the kernel. The argument the system calls takes is

  – `saadi_scheduler_t *param` The parameter to be filled with the scheduler information. The memory is to be allocated by the calling process.

- `long saadi_get_scheduler_of_class(int class, types_t *types);`
  Returns the scheduler types that are available of a certain specified class. The arguments the process takes are

  - `int class`: The class for which the types are to be returned.
  - `types_t *types`: This structure is filled with the available types. The memory is to be allocated by the calling process.

- `long saadi_get_scheduler_classes(classes_t *classes);`
  Returns the available scheduling classes. The argument the system call takes is
  - `classes_t *classes`: This structure is filled with the available classes. The memory is to be allocated by the calling process.

5.3.5 The SAADI Procfs Interface

by Markus Wuebben

The SAADI Procfs Interface has been deprecated. All functionality that was provided by the Procfs implemenation is now covered by the use of the SAADI system-calls (see Sec. 5.3.4). In addition a tool called `scontrol` is delivered which implements a set of convenience functions to access these system-calls.

6 Evaluation of results

by Tobias Malbrecht, Mattias Stöneberg (except 6.1)

In the previous chapters we described the function of SAADI. Now this chapter finally presents the results we got by running SAADI.
To see how SAADI performs, we first have to consider the input on which SAADI should be tested. The set of possible input is not bounded as well as not easy to formalize. Therefore, in section 6.1 we present some SDL examples, that SAADI can be tested with.
Since testing SAADI (i.e. running SAADIs genetic algorithm to find an optimal scheduler hierarchy for a certain SDL file) is a very time-consuming task [1], we have only run a few tests, whose results are shown in 6.2. This section also presents some measurements we made to compare SAADI and a standard Linux kernel, that we took as reference.
Finally, section 6.3 interprets the results.

6.1 SDL file examples

by Erdal Yigit

At this Time we use 5 different SDL files for the test environment. Every file has a different max. theoretical system load, number of applications, and various main focus from application types.

SDL example 1 A small SDL with 7 interactive, 4 periodic and 4 batch processes which approx. 65% system load.

SDL example 2 A overloaded SDL with 9 interactive, 8 periodic and 4 batch processes. The system load is approx. 200%.

SDL example 3 A SDL with 8 interactive, 5 periodic and 8 batch processes. This SDL describe a system with more interactivity programs like web browsers etc. The system load is approx. 95%

SDL example 4 This SDL describe a system environment with main focus on batch processes. The SDL specify 4 interactive, 5 periodic and 8 batch processes. The system load is approx. 115%

SDL example 5 A SDL which include the description for 4 interactive, 8 periodic and 4 batch processes. The main focus is in this time on the periodic processes, which become the most processor time. The system load is approx. 100%.

[1] Several minutes are needed to build, run and test a single individual (i.e. scheduler hierarchy).

6.2 Results

As shown in the last section, there are a lot of possibilties of scenarios in which the performance of SAADI can be evaluated. Here we pick one example scenario, whose results are show in this section. The sample SDL file taken represents a scenario in which the computer is highly overloaded. Under this assumption, it is likely, that most of the processes requirements could not be fulfilled. This probably would result in a relatively low fitness value, which can be computed from the difference of the processes requirements and the actually measured scheduling criteria (as outlined in chapter 2.4). In the following, we first concentrate on the results measured while running the genetic algorithm of SAADI.

6.2.1 Genetic algorithm of SAADI

Since it is unlikely, that the real optimum (i.e. a fitness value of 1) is actually reached, we limited the number of generations to 13. Figure 6.1 shows the progression of the minimum, maximum and average fitness values (per population and generation). As you can see from the curves, the fitness values range from a minimum of about 0.07 to a maximum of about 0.80. When we interpolate a linear function for each the minimum, maximum and average curves, we get three straight lines with each a slightly positive gradient. The greatest increase is in the minimum fitness values.

The most important value, which can be read off the chart, certainly is the maximum fitness value. This maximum is reached in the 10th generation. Its value is (as said above) about $0.80781419...$. The scheduler hierarchy of the corresponding individual is the shown in figure 6.2.

6.2.2 Linux standard kernel

To proof the great performance of the SAADI approach, we need a reference as a basis for comparison. The SAADI Linux kernel is based on the standard linux development kernel 2.5.58. It was a natural decision to choose this kernel as a reference for comparison with SAADI.
To get appropriate results, we of course applied the same input data as with SAADI, which means, we *run* the same SDL file (i.e. the same test applications). We run the test ten times and got values that range from 0.20 to 0.43.

6.3 Evaluation

Due to the wide application area and the resulting huge amount of input applicable to SAADI, we were not able to run sufficient tests. This section concludes performance and applicability from the test results described in the former section.

6.3.1 Genetic algorithm

As described in former sections, building an appropriate scheduler tree is a major task, which made the appliance of a genetic algorithm suggestive. The genetic algorithm applied in SAADI showed good performance, as can be seen from the results described in the last section. Relatively good performing individuals (i.e. scheduler hierarchies) are found throughout the process. Additionally, the not well performing scheduler hierarchies were assorted making the population more stable. As could be seen, generation 14 does not contain an indiviual with a fitness value less than 0.4. This outcome is especially acceptable when we regard the fitness values resulting from tests with the reference kernel.

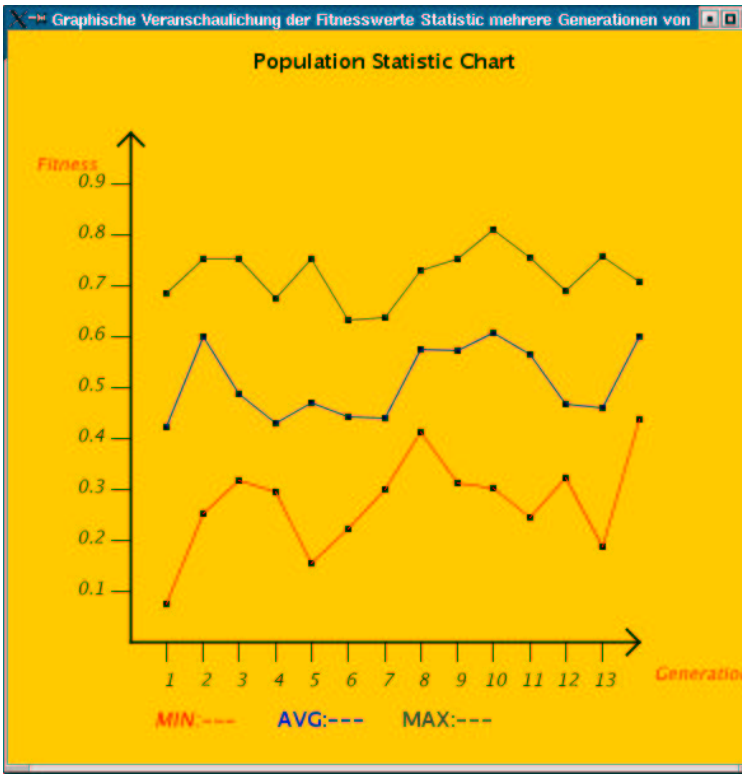


Figure 6.1: Maiximum, minimum and average fitness values of 12 generations.

6.3.2 Comparison to the standard linux kernel

As said above, the 14th generation of the SAADI genetic algorithm does not contain any individual with a fitness value less than 0.4. Especially, the overall maximum fitness value found was even more than twice that value. The fitness values resulting from tests with the standard linux developement kernel 2.5.58 we took as reference range had a maximum of only about 0.43. This shows the advantage of the SAADI approach over conventional general purpose scheduling. This result becomes overwhelming regarding the point, that these results were measured applying an SDL file dramatically overloading the computer. In contrast to best-effort scheduling, SAADI provides scheduling due to the application requirements. A fitness value of 0.8 as reached with a SAADI hierarchical scheduler running a set of applications highly overloading a computer shows that SAADI fulfills application needs in an adequate or even superior manner.

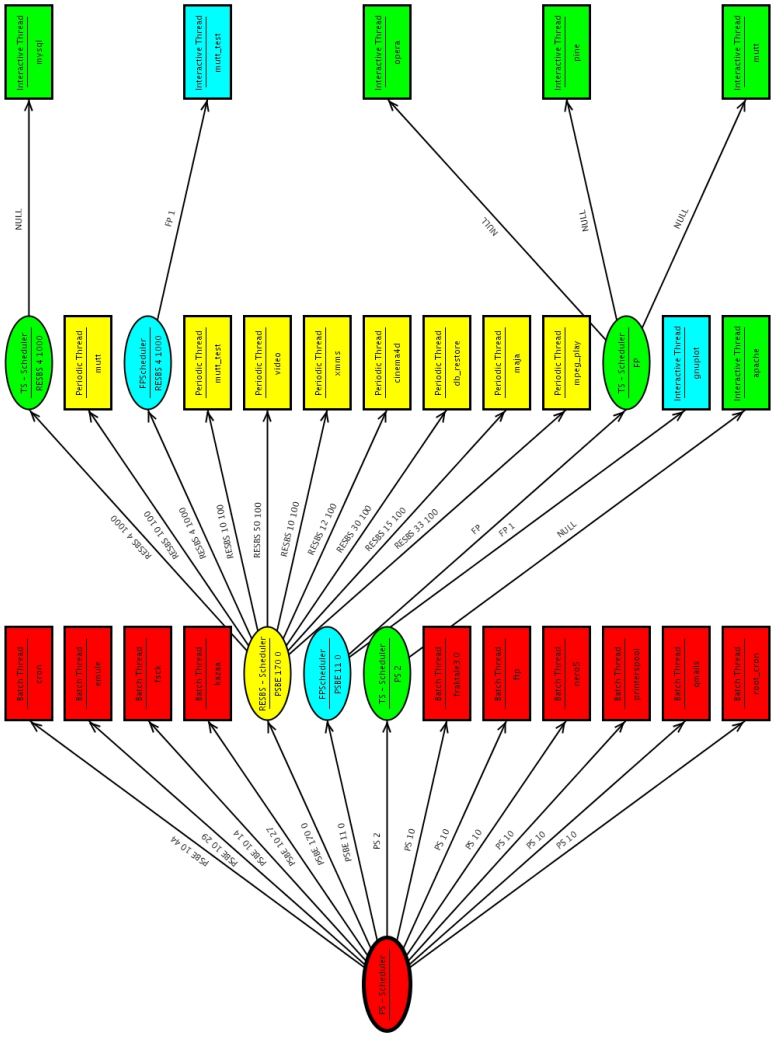


Figure 6.2: Best hierarchy in 12 generations.

7 Conclusion

by Mario Lischka

The most challenging aspects of this project were the integration of a framework for hierarchical schedulers into the Linux–kernel on the one hand, and modeling the scheduler hierarchy in such a way that genetic algorithms can be applied, on the other hand. In order to raise the general acceptance and usability of our approach we developed a graphical user interface for the SAADI Scheduler Description Language, which allows the user to specify her or his preferences for the scheduling. Another important aspect is the testing environment which interacts with those parts of SAADI that are responsible for generating different hierarchies, in order to find an optimal one for the given user requirements.

As we did not want to build a simulator to test our approach, but target real world application we choose the Linux–kernel to substitute the existing scheduler with a dynamic hierarchical scheduler. Additionally we can compare the original scheduler with our MHS and the results are very promising. It was a challenging task to integrate our framework into the current Linux–kernel, which is the interface between the scheduler–hierarchy and the other functions of the kernel and on the other side for various basic schedulers. In this project we have integrated basic set of schedulers which were used during our experiments. Further schedulers may be integrated easily for detailed description we may refer you to the Appendix A.2.

Another key aspects of this project is mapping guarantees to the scheduling hierarchies and a guarantee word which is used as input of genetic algorithms. The method presented in section 2.3 solves this problem and allows to include the restriction developed by Regehr [Regehr, 2001] and further extension (as described in the Appendix A.1).

As our approach is a *user centered scheduler design*, our GUI provides an easy to use interface for specifying the preferences of a user. These user requirements are expressed through constructs of the SAADI Scheduler Description Language, an XML based language, which allows to exchange of these information between the GUI, the genetic algorithms, the testing environment and future parts of this project in a standardized format. Future application may interact with the scheduler hierarchy of a kernel directly through system calls as described in section 5.3.4.

Finally we want to find an optimal scheduler hierarchy which fulfills all preferences given by the user. Therefore we developed a test environment which generates as yet artificial payload and determines the fitness of a given scheduler hierarchy. Based on these results several generations of hierarchies are tested in order to find an optimal one. Until now an optimal scheduling hierarchies were found only when the user requirements were not very challenging, but with harder user requirements our schedulers are performing far better than the original Linux $O(1)$ scheduler.

7.1 Future Works and Long Term Research

by Muddassar Farooq

SAADI has demonstrated that interesting and novel ideas from Evolutionary Algorithms could be incorporated in the real life operating systems. Due to lack of time and resources we could not completely explore different dimensions relating to our approach and hence interesting and challenging projects could be take in future to fully exploit the potential offered by SAADI.

- At the moment we generate offline hierarchies and then evaluate them with different test profiles to select the best one. However a more challenging and desirable feature is to extend the existing framework in such a manner that a hierarchy could be generated and modified on the fly during run time. This will not only simplify most of the task related to testing a hierarchy but also enable our scheduling system to really evolve in an adaptable fashion with changing dynamic environments.

- At the moment we have a very basic genetic algorithm in place just to show a proof of concepts model. Probably in future it makes sense to evaluate more advanced and sophisticated recombination, selection and mutation operators to conclude firmly about the real impact of evolutionary approach in the scheduling framework. Our preliminary findings are that in our framework genetic strategies do not really bring quantifiable benefits. However this is just a preliminary conclusion and needs to be verified with rigorous and extensive testing.

- In our work we did not consider the stolen time problem at all. Generally we assume that our scheduling system has 100 percent CPU time. However under greater network traffic, this is not true as the time to server hardware interrupts is taken from the scheduler in an operating system. Hardware interrupts have hight priority over normal scheduler. Probably an interesting task would be to study the behavior of stolen time and then try to approximate its behavior with a model so that in our ”guarantee” system we could take care of this important factor and the scheduling system is more robust.

- At the moment a user specifies the requirement of tasks himself in the SDL file. An interesting approach would be to develop an analyzer tool that could parameterize different characteristics of an application and then determine its scheduling requirements.

- SAADI could be extended to run on multiprocessor systems. An interesting approach to investigate would be to run on each processor a specialized scheduler and then these group of schedulers could cooperate with each another to schedule different tasks. Quite interestingly then this cooperating could be seen as Multi-Agent System where different agents try to work together through communication and cooperation to achieve an overall optimum for the complete system.

- SAADI at the moment do not consider hardware architecture of the machine on which it runs. In future we could extend SAADI framework so that it tries to adapt to architectures like Pentium Speedstep that has got a Dynamic Voltage Scaling (DVS) feature. All schedulers could convey their idle time information to the root scheduler who could then make an informed decision whether to downscale voltage or frequency of the processor. If schedulers are implemented like agents then root scheduler could negotiate different energy saving options with its children e.g.

We could save 10 percent energy as a system in case you are willing to accept a 20 msecond response time instead of 15 msecond and simulate interesting options.

- In SAADI framework developers have to implement their won scheduler to incorporate them. Probably yet interesting approach would be to develop another framework which helps in writing interesting simple new schedulers from interesting scheduling policies.

A Extension of SAADI

In the following we describe the procedure, how to extend SAADI concerning the abstract scheduler model as well as the Linux kernel provided with SAADI. The possibility of extending SAADI due to further requirements is made possible by the high grade of modularitiy in the implementation of SAADI.

A.1 ASM

by Simon Muras

We will now discuss the possibility of adding new scheduler and eventually corresponding guarantees to the *Meta Hierarchical Scheduler*. For a new scheduler type δ and a corresponding new guarantee type γ two classes *edu.udo.saadi.asm.*δ*Scheduler* as subclass of *edu.udo.saadi.asm.Scheduler* and *edu.udo.saadi.asm.*γ*Guarantee* as subclass of *edu.udo.saadi.asm.Guarantee* have to be written. You can of course choose existing subclasses like *edu.udo.saadi.asm.RESScheduler* as superclass if you want to realize a $RESBH$-scheduler.
The next step is the modification of the given grammar file for the MSG. Take notice that both guarantee productions (like $\gamma \rightarrow \ldots$) and scheduler productions (like $SCHED - \delta - * \rightarrow \ldots$) need to be named like the given scheduler and guarantee classes.
If a scheduler of type δ needs in addition system resources (e.g. some sort of $share$ like the PS-scheduler the method $evaluateParent$ of the class $Hierarchy$ needs to be rewritten.
Last but not least the addition of a new scheduler to the theoretical framework the code generation process needs to be slightly modified, means the scheduler-$c - code$ mapping must be realized, for a detailed description see chapter.

A.2 Kernel

by Piotr, Rolf

To implement a new scheduler named δ, a new file named saadi_sched_δ.c has to be created in the kernel/ directory of the kernel source. In this file the functions join, leave, link, unlink, schedule, construct and destruct have to be implemented as described in 5.3.2 the scheduler specific data (the runqueue etc.), the struct runqueue, defined in include/linux/saadi.h, contains the union impl. The new scheduler can use a member of this union or a new data struct can be defined. Also in saadi.h the schedulable specific data is defined in the struct schedulable. The union sched_data can be extended to contain the data for the new scheduler.

In include/linux/saadi_sched_types.h the new scheduler type SCHED_δ and if it is of a new guarantee type scheduler class also the SCHED_CLASS_γ has to be defined.

Now the kernel/Makefile has to be modified to include the new sourcefile in the files to compile for the kernel. In the line obj-y= ... our saadi_sched_δ.o has to be added and voila the next saadi run can use the new scheduler.

B Bibliography

[Earley, 1970] Earley, J. (1970). An efficient context-free parsing-algorithm. pages 451–455.

[PG Saadi, 2003] PG Saadi (2003). Interim Report of Project-Group 424 “Saadi - Integrated Approach for Adaptable Schedulers”. Interne Berichte, Universität Dortmund, Fachbereich Informatik.

[Regehr, 2001] Regehr, J. (2001). Using hierarchical scheduling to support soft real-time applications on general-purpose operating systems.

[Stolcke, 1995] Stolcke, A. (1995). An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. In *Computational Linguistics, (MIT) Press for the Association for Computional Linguistics*, volume 21.