Specification and Analysis of Power-Managed Systems

ALESSANDRO BOGLIOLO, LUCA BENINI, EMANUELE LATTANZI, AND GIOVANNI DE MICHELI, FELLOW, IEEE

Contributed Paper

Dynamic power management encompasses several techniques for reducing energy dissipation in electronic systems by selective slowdown or shutdown of components. We present a theoretical framework for explaining and classifying different approaches to power management. Within this framework, we model power-manageable components, workloads, and controllers as discrete-event systems (DESs). The structure of these DESs is specified in terms of physical states (representing operation modes) and events (triggering state transitions), while system behavior is specified in terms of next-event and next-state functions. In particular, nondeterministic next-event and next-state functions are modeled by conditional probability distributions, according to generalized semi-Markov processes (GSMPs).

The modeling framework provides a general denotational model for system specification and a rigorous execution semantics that enables event-driven simulation. We introduce a modeling framework, built on top of MathWork's Simulink, supporting the specification and execution of our model. In particular, we present templates for the Simulink simulator to execute GSMP models, and we describe how to use such templates for specifying, analyzing, and optimizing dynamic power-managed systems.

Finally, we demonstrate the expressive power and versatility of the proposed approach by using the modeling framework and the simulator for the analysis of representative real-life case studies, including the Intel Xscale processor architecture, a multitasking real-time system, and a sensor network.

Keywords**—***Low-energy design, power management, stochastic control, system on a chip.*

I. INTRODUCTION

The average citizen of an industrialized country interacts with tens of electronic appliances every day, while working, traveling, and at home. Arguably, the diffusion of electronics is one of the distinctive characteristics of human civilization in the new millennium. All electronic systems consume energy to perform their task and the impact of electronic systems on the power budget of densely populated areas is steadily growing. At the same time, portable, embedded electronics are becoming increasingly pervasive in today's life: cellular phones, electronic organizers, and digital cameras are just a few examples of devices that are considered almost indispensable for work and entertainment. Reducing power consumption in these devices enables aggressive miniaturization, longer time between battery recharges (or changes), and, ultimately, deeper market penetration.

Energy efficiency is a primary design objective for increasingly large classes of electronic systems. While in the past energy efficiency was almost invariably associated with battery-powered operation [20], [21] nowadays it has become a concern also for electric-grid powered equipment, like servers, network switches, routers, etc. [22]. The quest for energy efficiency has two main facets. On one hand, new systems should be designed with clearly specified energy budgets, and energy consumption considerations should percolate through all phases of the design process, from system conception to elementary devices and technology. On the other hand, electronic systems should be managed efficiently from the power viewpoint during their in-field operation. The focus of this work is on power management of electronic systems.

The fundamental rationale for power management is quite intuitive. Systems are generally designed to deliver specified levels of performance under heavy load. During in-field operation, a system is often underutilized and, therefore, it can be operated in a reduced performance mode to save energy. If the entire system, or some of its components, are completely idle, they can be shut down while waiting for some service request. In the past, power management has been more an afterthought than a fully developed design discipline. Unfortunately, when a system has not been designed with energy ef-

Manuscript received December 4, 2003; revised April 12, 2004. This work was supported by the National Science Foundation under Grant CCR-0305718.

A. Bogliolo and E. Lattanzi are with the Information Science and Technology Institute, University of Urbino, Urbino 61029, Italy (e-mail: bogliolo@sti.uniurb.it).

L. Benini is with the Department of Electronics and Computer Science, University of Bologna, Bologna 40127, Italy.

G. De Micheli is with Stanford University, Stanford, CA 94305 USA.

Digital Object Identifier 10.1109/JPROC.2004.831207

0018-9219/04$20.00 © 2004 IEEE

ficiency objectives, transitions from a fully operational state to low-power, reduced functionality modes of operation are slow and expensive, or, in some cases, simply not possible. Hence, the *power manager* (PM), i.e., the system entity in charge of controlling mode transitions, has limited degrees of freedom. Furthermore, it should carefully weight the costs of mode transitions (which are certain when a transition is made) against the potential energy savings (which depend on the uncertain duration of the idle condition).

Power management strategies (also called *policies*) in the above outlined setting have been studied intensively in the last few years [23]. Substantial energy savings have been achieved when designers have started supporting power management. Recently, the electronic component market has witnessed the announcement of numerous new components with multiple, finely controllable modes of operation that trade off performance with power consumption, e.g., variable-frequency, variable-voltage operation [24], [25]. Components with multiple active states based on dynamic frequency and voltage scaling have created many new opportunities for dynamic power management (DPM). In such a rapidly evolving setting, the modeling, analysis, and optimization techniques developed in the past are not general and powerful enough, and a richer theoretical framework is needed to sustain the development of novel, advanced power management techniques.

The main objective of this paper is to build such a framework in a theoretically sound fashion, leveraging abstractions and techniques from several correlated disciplines, such as stochastic modeling, discrete-event systems (DESs), simulation theory and sensitivity analysis, stochastic optimization, and control. We aim at bridging the gap between electronic system designers and researchers working in the above mentioned fields, in order to facilitate cross fertilization and multidisciplinary research in a strategic area for current and future applications.

The paper is divided in four parts. First, a general overview is given of technology and architectural trends. The main purpose of this section is to demonstrate the increasing importance of DPM and to contextualize our work. The remaining three sections are the technical core of the paper, focusing on system modeling, simulation, and applications. We will model power-manageable components, workloads, and controllers as DESs [26]. The structure of a DES is specified in terms of physical states (representing operation modes) and events (triggering state transitions), while system behavior is specified in terms of next-event and next-state functions. The modeling framework we propose provides: 1) a general denotational model for system specification (supporting, in particular, composition and abstraction); 2) a rigorous execution semantics that enables event-driven simulation; and 3) a formalism for specifying the probabilistic structure of *generalized semi-Markov processes* (GSMPs) [27]. The modeling framework provides us with a means of classifying different DPM systems based on the properties of their models.

Next, we present a system modeling infrastructure, built on top of MathWork's Simulink [28], supporting the specification and execution of DES/GSMP models. In particular, we describe templates for the Simulink simulator to execute GSMP models, and we describe how to use such templates for specifying DPM systems of practical interest. Finally we show how to use the simulator to evaluate and optimize system parameters and DPM policies.

In the last part of the paper, we demonstrate the expressive power and versatility of the proposed approach by using the modeling framework and the simulator for the analysis of representative case studies.

II. Power Management: Why and How

As outlined in the introduction, system designers are faced by an unprecedented *power crisis*, and power management is currently the most effective response to the challenge. To fully motivate this assertion, we first review the trends of evolution of current technologies and architectures, then we analyze the state of the art in power-manageable systems (PMSs) and give a few representative examples.

A. Technology Trends

The trends in semiconductor technologies are characterized by decreasing feature sizes and increasing device densities. As a result, the energy dissipated per unit area is rising and is posing an unprecedented challenge to designers. To cope with this and other problems (e.g., hot carrier effects), the supply voltage is also reduced. Nevertheless, the downscaling of supply voltages is not sufficient to contrast the increasing power consumption trends in chips.

Indeed, there are three factors that contribute to increasingly higher energy dissipation with downscaled technologies: the energy on global interconnect wires, the aggressive increase of operating frequencies and the dominating contribution of leakage currents. We analyze these three factors next.

First, silicon technology is becoming increasingly interconnect dominated. Since global wire length does not scale down, both delay and energy dissipation on global interconnect dwarf those of computation and storage units. Indeed, while the gate capacitance of minimum-size transistors is decreasing, the interconnect capacitance per unit length is not decreasing at the same speed (because of fringing capacitance contributions) and interconnect length is not decreasing for global wires. Hence, the C factor in the well-known switching power equation $P_{\rm sw} = kCV_{\rm DD}^2 f$ (where k represents switching activity) does not scale down as fast as minimum feature size [29], [30].

Second, in order to reap performance benefits (in other words, to satisfy quality of service requirements), chip clock frequency is scaled faster than technology [31], i.e., the percentage increase in frequency in an upgraded technology is higher than the percentage decrease of feature sizes. This result is achieved by clever architectural optimizations that reduce the number of logic stages to be traversed within the clock cycle time. From a power viewpoint, this is clearly a problem, because power is directly proportional to switching frequency f. Performance constraints (or objectives) are also the main reason why supply voltage scaling is not as drastic

as one would desire for power minimization purposes. In fact, transistor switching speed decreases as $(V_{\mathrm{DD}} - V_T)^\alpha$, with $1 < \alpha < 2$. This cannot be tolerated if performance is tightly constrained.

Third, deep submicrometer transistors are increasingly leaky in the OFF state. Source-to-drain current due to subthreshold conduction is the dominant cause of leakage, but drain-to-gate currents due to electron tunneling across the gate oxide is also becoming significant. Furthermore, random variations of the number of dopant atoms in the channel region cause poor threshold control, and many transistors have a threshold significantly lower than nominal. Unfortunately, subthreshold conduction is exponentially dependent on threshold voltage and transistors with lower threshold leak exponentially more than nominal transistors. As a result, chip standby power is becoming a significant concern. Clearly, subthreshold leakage is also a heavy limiter to threshold voltage V_T reduction, with obvious negative impact on supply voltage scaling [31].

In summary, power dissipation will grow significantly in future technologies, unless innovative design techniques are used. Because of the increasing importance of leakage currents, even standby power is expected to increase. Technologists are pointing at power as the one of the most likely show-stoppers to technology scaling if adequate countermeasures are not taken.

B. Architectural Trends

Technology evolution is not going to solve the power consumption problem. On the contrary, many technologists refer to design innovation at the circuit, logic, architectural level as a way out from the crisis. Unfortunately, trends in this area are not favorable to energy efficiency at all. Even though a significant research effort is being devoted to power minimization, mainstream architectural design is moving toward energy-hungry architectures.

Most electronic systems are nowadays designed with a high degree of programmability. The majority contains one or more core processors, and many instantiate several programmable coprocessors (e.g., very long instruction word (VLIW) units for numerical computations, programmable IO processors, etc.). A few recent designs even embed a significant amount of bit-programmable logic (field-programmable gate array (FPGA) fabrics) [32]. Programmability is a fundamental requirement when designing large-scale systems on chips (SoCs) for three main reasons. First, it ensures functional flexibility, which widens the spectrum of applicability and the potential production volume. Second, it leaves margins for postfabrication bug fixing and tuning, thereby enhancing yield. High yield and volume of sales are required to amortize ballooning mask development and fabrication costs. Third, processor-based architectures reduce design time because they emphasize reuse of hardware modules (e.g., the cores themselves, the memories, etc.) as well as software components (e.g., libraries, operating systems, compilers).

Yet, flexibility and reuse come at a price. The power–performance ratio (i.e., the energy) required by a processor to carry out a given task (e.g., MPEG decoding) is several orders of magnitude (three or more) higher than what could be achieved with an application-specific architecture [21], [33]. Advanced processors, often required to attain performance goals, are even more power-hungry than simple processors, because they rely on various forms of speculative execution to increase the average number of instructions executed in a clock cycle. Well-known performance enhancement techniques, such as speculating past branches, value prediction, and prefetching, imply the execution of redundant operations which increase power consumption.

Fine-grained programmable fabrics can be one or more orders of magnitude more energy efficient than processors [33] for some classes of computations, but they still incur a very significant overhead with respect to dedicated logic. Recent data shows that computation performed by an embedded FPGA fabric in a hybrid FPGA–application-specific IC (ASIC) chip is more than two orders of magnitude more power consuming and more than ten times slower than the same computation in dedicated logic [32]. The overhead in this case is mainly due to communication between fine-grain programmable logic elements, which is performed on massively redundant programmable wiring resembling a multistage network, as opposed to dedicated, instance-specific wires. Additionally, programmable logic blocks are often much more complex than the basic gates they mimic when programmed.

C. Power-Manageable Hardware

As seen above, programmability is generally adverse to power efficiency. However, there is a way to profitably exploit programmability to reduce power consumption, namely, via power management. Supporting power management requires adding hardware resources that do not have a computational task, but they dedicated to controlling the power level of functional units, detecting and exploiting idleness, and locally trading off performance with power. Power-manageable architectures ultimately aim at reducing idle power (i.e., the power consumed by a hardware component when it is not in use) and active power, through the dynamic control of transistor thresholds and supply voltage, clock activity, and frequency.

To reduce leakage power in idle state, variable-threshold circuits control the threshold voltage of transistors through substrate biasing. When a variable-threshold circuit becomes quiescent, the substrate of NMOS transistors is negatively biased, and their threshold increases because of the well-known *body-bias effect*. A similar approach can be taken for PMOS transistors (which require positive body bias). Variable-threshold circuits can in principle solve the quiescent leakage problem, but they require standby control circuits that modulate substrate voltage [34].

Idle power is not only caused by leakage, but it is also due to unneeded switching activity. Clock switching within idle functional units is the best-known example of this problem. Clock gating [35] is used to eliminate unneeded clock activity. Most low-power processors implement both

hardware and software controlled clock gating through dedicated power-down instructions. A radical way to eliminate idle power (both leakage and dynamic) is to disconnect a unit from its power supply [20]. Unfortunately, in this case state information is lost; therefore, relevant state bits must be saved (for instance, in nonvolatile memories) before shutdown and recovered upon restart. State saving and restoring significantly increases the overhead associated with shutdown transitions.

To minimize active power, circuits should be run as slow as possible (within performance constraints) to eliminate slack while at the same time minimizing supply voltage [20]. This translates into cubic power savings (and quadratic energy savings). Variable-voltage circuits [24], [36] offer the possibility of modulating the power supply dynamically during system operation. In practice, the implementation of this technique requires considerable design ingenuity. First, voltage changes require nonnegligible time, because of the large time constants of power supply circuits. Second, the clock speed must be varied consistently with the varying speed of the core logic when supply voltage is changed. Even though slowdown coupled with voltage reduction is more effective than running the circuit at maximum speed and then shutting it down as soon as it becomes idle [36], the two approaches are not mutually exclusive. Clearly, slowing down a circuit reduces its idleness, but in many cases it cannot eliminate it.[1]

Power-manageable hardware can be abstracted as a state machine where states represents various modes of operation and transitions have a cost [23]. In the remainder of this section, we give a few examples of complex, practical PMSs, and we describe informally their state-based representations. The same examples will be formally described and analyzed in a quantitative fashion in Section V.

1) Multiple Power States: Modern power-manageable processors support multiple shutdown modes, as well as variable-voltage operation. The Intel Xscale processor is an example of an architecture with advanced power management features, namely: 1) user controllable processor speed and voltage supply; 2) multiple sleep states; and 3) performance monitoring hardware. The first prototype Xscale processor [37] could run with core voltage ranging from 0.70 to 1.65 V, the corresponding clock frequencies being 50 and 800 MHz, respectively. At top speed, power consumption is 900 mW and power efficiency is 850 million instructions per second per watt (MIPS/W). At the lowest speed, power consumption is 55 mW, with a corresponding power efficiency of 4500 MIPS/W. Supply voltages could be varied without stopping or resetting the processor, with a maximum slew rate of 4 mV/μs. However, clock frequency changes require 20 μs. to relock and stabilize the clock generator. Three inactive states are supported, namely, *idle*, *drowsy*, and *sleep*. In idle state, the clock distribution is gated off, but the clock generator keeps running. Power is reduced to approximately 10 mW, and exit from the idle state requires a single clock cycle. In drowsy, or standby mode, the phase-locked loop (PLL) clock generator is turned off, reducing power consumption to 100 μW. Exit from the drowsy state requires approximately 20 μs. Finally, in sleep mode, the core is completely powered down, reducing power virtually to zero (with the exception of pin power), and internal state is lost. Resuming operation from sleep requires a complete processor reset sequence and context recovery (several thousand cycles).

The first commercial component derived from the Xscale architecture, the Intel 80 200 core processor, is very similar to the research prototype described above, with a few exceptions. First, clock speed changes are quantized. Clock speed is set by writing to a special control register. Ten different speeds are supported and lock speed changes require approximately 1000 clock cycles. Even though the processor could in principle run at ten different supply voltages, one for each available clock speed, a maximum of four voltage levels is recommended. Hence, we have two different speeds for each supply voltage. Furthermore, the *drowsy* state is not supported (only idle and sleep are available). Exit from idle takes approximately ten cycles, and resuming from sleep requires a few milliseconds, depending of the amount of state information that must be saved and restored. Probably the most important extra feature available in the commercial component is a large number of hardware monitoring registers that provide accurate counts of many significant runtime parameters, such as cache misses, table look-aside buffer (TLB) misses, exceptions, etc. These counters can be very useful in determining actual processor performance at runtime, thereby facilitating the implementation of "closed-loop" policies, where power management decisions are based on runtime performance estimates [15].

Xscale is only a single design point. Several variable-voltage processors have been announced, and some are available on the market [24]. Furthermore, with the diffusion of multiprocessor SoCs (MPSoCs) for embedded multimedia and signal processing applications, the number of variable-frequency/variable-voltage cores integrated onto a single chip is going to increase very rapidly. Thus, there are ample and growing opportunities for DPM schemes exploiting the additional degrees of freedom offered by variable-voltage operation.

Variable-voltage processors are not the only electronic components supporting multiple active and sleep states. Wireless network interfaces are another important family of devices with similar characteristics. For the sake of illustration, let us focus on IEEE802.11b (WIFI) cards [16], [17]. WIFI cards have multiple inactive states, whose precise definition is implementation and vendor specific. In general, most cards support an "off" state, where power dissipation is negligible and association with the wireless network is lost. A card in this state is not responsive and cannot be located by the base station. A transition out of the off state is very time-consuming (a few hundreds of milliseconds), mainly because it involves not only hardware activation, but also network reassociation. Other inactive states ("doze") are

[1]Consider, for instance, an MP3 player. When the device is active, it operates under tight performance constraints (namely, real-time playback), and obviously it is not possible to stretch execution time beyond the duration of a music track, even if after it has been played out, the player remains idle for hours.

supported, where the card does not lose association with the network, but it does not explicitly transmit or receive data. This behavior is enabled by a feature of the IEEE802.11 protocol family, which provides for periodic synchronization frames (beacons). When idle, the card is activated very briefly only to deal with beacons. Doze states have small reactivation time of a few milliseconds and do not imply any packet loss (thanks to buffering at the base station). Power consumption is a factor of five to ten lower than that in receive state, but it is not negligible (in the order of tens of milliwatts).

The most power consuming state for WIFI cards is the transmit state. When transmitting, the output RF amplifier is active, and significant power is radiated through the antenna, in addition to the power consumed for baseband processing. Power consumption in this state can exceed 1 W, a factor of two higher than the receive power. Advanced cards support output power control to allow fine tuning of transmit power consumption to environment and network conditions. Output power can usually be selected from a finite set of available values; transitions among power levels are fast (a few milliseconds).

Clearly, wireless network interface cards are very flexible in terms of available power states. Optimal power management for these devices is a complex task, which is often tackled with distributed policies, as discussed in the next section.

2) Interacting Power-Manageable Devices: Complex systems can contain multiple power-manageable devices. In this case, power management requires fine-grain decision on state transitions for multiple interacting devices. If devices are closely coupled (e.g., on the same die or board), power management can be performed either by a single centralized controller or by local interacting controllers [56]. As coupling becomes weaker, centralized power management becomes increasingly hard, mainly because is difficult and expensive to collect information and issue commands to many distributed, loosely coupled components.

Wireless networks are perhaps the most well-understood example of distributed PMSs. A huge amount of research has focused on power management for wireless networks, especially for widespread technologies like IEEE802.11a/b [5], [7], [12], [14]. Even though most of the seminal work in this area was theoretical in nature, it has propelled the development of advanced network cards (like the ones described in the previous section) with a high level of power controllability. Thus, many approaches developed in the past are likely to find practical application in the near future. We can coarsely classify power management schemes in two broad classes: those focusing on transmit power reduction [13], and those aiming at minimizing the power spent in receive mode [2], [11], by intelligent exploitation of low-power inactive states (like the “doze” and “off” states described in the previous section). The first class of techniques is particularly relevant in *ad hoc* configurations, where base stations are very sparse or absent, and power-constrained clients have to transmit significant amount of data because of node-to-node forwarding. The second class of techniques has general applicability, since the percentage of idle time for wireless network interfaces is significant for many application scenarios.

Power management policies have been developed at various levels of the network stack, starting from the medium access control (MAC) layer [8], [9] up to the application layer [1], [10]. In general, the proposed techniques imply some form of coordination among multiple network cards and/or the base station, which requires some extra communication. Recurring themes are: 1) minimization of the extra traffic caused by power management (distributed and weakly coordinated policies [4]) and 2) exploration of the computation and storage versus communication power tradeoff (e.g., data compression or caching to reduce the amount of data to be communicated [6], buffering to enable network shutdown [3]).

An extreme example of loosely coupled PMS is a wireless *sensor network* [19], [38]. Sensor networks have gained importance in numerous civil and military applications. They can be used to continuously monitor the environment for various types of events, and they operate in a reconfigurable and adaptable fashion under extremely tight power constraints [39]. The node of a sensor network consists of one or more embedded sensors, analog-to-digital converters, a processor with memory, and an RF section for communication with other nodes. Each node is power manageable. Nodes are distributed in a target area often in an irregular fashion and they are usually required to have reduced size (a few cubic centimeters). They are battery operated or self-powered (with a variety of energy-scavenging techniques).

The first practical sensor networks are emerging from research laboratories. A large-scale sensor network with 800 tiny nodes (the size of a quarter), has been recently demonstrated [40]. The node contain a 4 MHz low-power microcontroller (ATMEGA 163) providing 16 KB of flash instruction memory, 512 B of static RAM, analog–digital converters (ADCs), and various simple peripheral interfaces. A 256-KB electrically erasable programmable ROM serves as secondary storage. Sensors, actuators, and a radio network serve as the I/O subsystem. The network utilizes a low-power radio (RF Monolithics T1000) operating at 10 kb/s. The node contains four types of sensors: light, temperature, battery level, and radio signal strength. It can actuate two LEDs, control the signal strength of the radio, and transmit and receive signals. By adjusting the signal strength, the radio cell size can be varied from a couple of feet to tens of meters, depending on the physical environment. A second microcontroller is provided to allow all cores to be reprogrammed over the network. The entire system consumes about 5 mA when active. The radio and the microcontroller consume as much power as a LED. In passive (sensing) mode, they consume only a few microamps while still checking for radio or sensor stimuli that can wake them up.

Power management for networks, primarily targeting sensor networks, is a rapidly developing research area, which spans all layers of the communication protocol stack [41], as well as operating systems and hardware abstraction layers [40], [42]. Needless to say, there are ample opportu-

nities for reducing power consumption in sensor networks, which are characterized by high redundancy and extreme levels of node idleness [18] (nodes are often idle for 99% of the operation time). One of the critical challenges in this area is to minimize idle receive power while at the same time maintaining adequate levels of responsiveness. Another key open issue is the definition of aggregate quality of service and energy efficiency metrics for an entire network, which can be quite different, and often in contrast with single-node metrics.

3) Energy Sources: In many battery-operated mobile applications the ultimate objective of power minimization is ensuring long battery lifetime. It has been shown in [23] that average power reduction and battery lifetime extension may be numerically far apart. This implies that optimizations for minimum average power may not be equally effective in extending battery lifetime, and vice versa.

Charge storage in a battery can be modeled as a capacitor with capacitance $C = 3600 \cdot \mathrm{CAP}$, where CAP is the nominal capacity in ampere $\cdot$ hours, which is usually provided in the battery's data sheet. By setting the initial voltage across the capacitor $V_C = 1$, we initialize the battery to its fully charged state. Unfortunately, the simple linear capacitor model is not accurate enough to model complex phenomena observed during battery discharge. In fact, the following three major effects must be taken into account.

- Battery voltage depends nonlinearly on its state of charge: voltage V_{Batt} decreases monotonically as the battery is discharged, but the rate of decrease is not constant.
- The actual usable capacity of a battery cell depends on the discharge rate: at higher rates, the cell is less efficient at converting its chemically stored energy into available electrical energy.
- The "frequency" of the discharge current affects the amount of charge the battery can deliver: the battery does not react instantaneously to load changes, but it shows considerable inertia, caused by the large time constants typical of electrochemical phenomena.
- Batteries operated at high discharge rate for a short period can recover available charge if the current load is temporarily reduced.

Various approaches have been proposed [43] to model nonideal batteries. An interesting state-based model has been formulated by Chiasserini and Rao [44]. A battery is modeled as a finite-state system, where states represent various charge conditions of the battery, characterized by different voltages. Transitions are caused by current load (to model dependency of discharge rate from load current). The recovery effect can be modeled as a nonzero transition probability toward higher charge states when load current drops to zero. The low-pass filtering effect can be modeled taking the running average of current load.

D. A Fragmented Landscape

Concluding this section, observe that the examples we described are representative of large classes of PMSs, and they demonstrate the variety of embodiments of power management problems in practice. Even though researchers have devised effective techniques in many of these areas, the whole field is characterized by significant fragmentation, which often leads to pitfalls (such as unrealistic assumptions and lack of experimental validation on real-life PMSs) and multiple rediscoveries of the same concepts and techniques.

One of the most striking examples of fragmentation is between shutdown-based and variable-voltage-based power management [also called *dynamic voltage scaling* (DVS)]. Two largely nonoverlapping groups of researchers have worked on the two themes in the past (refer to [45], [46] for an overview of DVS and to [23] for a survey on shutdown-based DPM), and many have also claimed the superiority of one approach over the other. However, shutdown and voltage scaling are two facets of the same problem: a power-manageable device can have multiple sleep states and multiple active states, characterized by different supply voltage and clock frequency values. Devising an effective power management policy for such a device requires deciding not only transitions between multiple voltages but also when to shut down the device and into which sleep states. Decoupling the two problems can only lead to suboptimal solutions.

Furthermore, many new power management problems are emerging. For instance, several researchers have recently focused on leakage power reduction, which has become a serious concern for large memories in deep submicrometer technologies [47]–[49]. Memory leakage power can be reduced by transitioning unused memory banks to a sleep state from which they can be reactivated with some extra penalty (in terms of activation time and/or stored content loss). This is a power state transition cost versus benefit, which is the core issue in DPM policies and algorithms. It is then possible (and desirable) to leverage the large body of knowledge developed in the area of shutdown-based power management to devise effective memory leakage reduction techniques.

These two examples provide additional evidence of the need for unification of the many flavors of DPM under a common formal modeling framework. This is the main objective of the following sections.

III. MODELING

Abstract models are required to formally analyze the properties of physical systems. In the case of power-managed systems, the main challenge is to strike the balance between a high level of detail, including functionality and focusing exclusively on power consumption models. As seen in the previous section, real-life power-manageable components are characterized by a number of different states of operation (power states), trading off power for performance and/or reactivation latency. Hence, a state-based model appears to be a natural abstraction. Furthermore, power state transitions are triggered in an enumerable set of time instants. Therefore, the most natural formal framework for studying the evolution in time of these systems is a DES model [50].

DESs have been extensively studied in the operation research and control systems communities, where DES simulation and simulation-based optimization are among the most widely adopted analysis and optimization approaches [51]. One of the distinctive characteristics of DES models is the presence of sources of nondeterminism (i.e., random events) that represent both modeling uncertainty and the uncertainty caused by the environment (e.g., the workload). For our specific application domain, we wanted to specialize the highly generic DES model in an effort to emphasize the following key characteristics.

- State-based component models, which naturally match the characteristics of real-life power managed components.
- Formally defined composition rules to allow hierarchical composition of complex power managed systems from simpler ones without losing the properties of the model.
- Formally defined sources of nondeterminism, to clearly decouple the parts of the system that are deterministically characterized from the sources of uncertainty.

Among many state-based stochastic DES formalisms, we chose to model PMSs as GSMPs, applied to DESs by Glynn in 1989 [27]. As detailed in the following sections, GSMPs satisfy all the requirements listed above. On the other hand, in contrast with less expressive state-based stochastic models (e.g., Markov or semi-Markov processes), GSMPs are general and powerful enough to model realistic power managed systems. Moreover, GSMPs provide both a rigorous denotational framework that drives system taxonomy and formal analysis, and an executable semantics compatible with event-driven simulation [52].

In the rest of this section, we first introduce a denotational model, a compositional rule, and an execution semantic for generic DESs, then we specialize our model for GSMPs. The combination of DESs with GSMPs is the major contribution of this section, and it provides a framework for analyzing a power-managed system from a formal standpoint.

A. Denotational Model

We denote by S the set of *physical states* (i.e., operation modes) of a system and by E the set of *events* that can trigger state transitions. All events compete to trigger state transitions: at each state, the event that comes first wins and is called the *triggering event*. The arrival times of triggering events are called *decision epochs*, since they are the time instants at which next states are chosen. State transitions are instantaneous.

We call *annotated physical state* a triple (s, λ, e), where s is the physical state, λ the decision epoch at which s was entered, and e the event that triggered the transition. The sequence of annotated physical states visited by the system up to the nth decision epoch is its *physical trajectory* at n, denoted by x_n. In general, we use subscript n to denote the nth decision epoch (λ_n) and all quantities referred to that epoch. The set of all possible n-step trajectories is denoted by X_n.

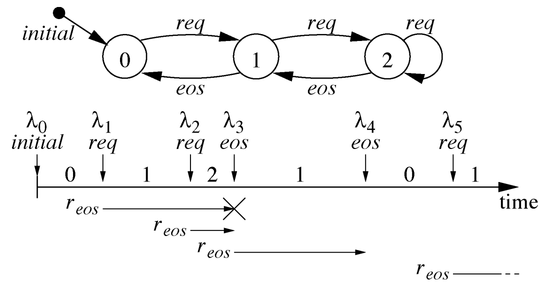


Fig. 1. State diagram and possible trajectory of a server with queue of length 2.

Clearly, $x_n \in X_n$. At decision epoch λ_0, the only feasible trajectory is empty: $x_0 = \varnothing$.

At decision epoch λ_{n+1}, the next state (s_{n+1}) depends on the trajectory at time n, on current time (λ_{n+1}) and on the triggering event (e_{n+1}). We call *initial state* s_0 the state entered at decision epoch $\lambda_0 = 0$ upon arrival of an implicit *initial* event.

We call the *next-state function* a black-box function F taking as input a trajectory, a decision epoch, and a triggering event and returning a target next state

$$s_{n+1} = F(x_n, \lambda_{n+1}, e_{n+1}).$$

According to our assumptions, if the input trajectory is empty, the decision epoch is $\lambda_0 = 0$ and the triggering event is $e_0 = initial$, then the next-state function returns the initial state s_0

$$s_0 = F(\varnothing, 0, initial).$$

In general, state transitions can be triggered by both *internal* and *external* events. We denoted by E_I and E_E the sets of internal and external events, respectively ($E = E_I \cup E_E$). While the system has no direct control on external events, it directly generates internal events depending on its history, represented by the trajectory at current decision epoch λ_n. We call the *next-event function* a black-box function G that takes in input the current trajectory x_n and returns a candidate triggering event $e \in E_I$ together with its residual time r_e.

$$(e, r_e) = G(x_n).$$

Function G is reevaluated at each decision epoch. If no external event arrives between λ_n and $\lambda_n + r_e$, then e will be the next triggering event ($e_{n+1} = e$) at decision epoch $\lambda_{n+1} = \lambda_n + r_e$. Otherwise, the state transition will be triggered by the external event at decision epoch $\lambda_{n+1} < \lambda_n + r_e$, and event e will be canceled. If no internal events are defined, or the system has entered a waiting state where it has to stay until a given external event occurs, then G returns $(none, \infty)$.

Example 1: Consider a server with a queue of length 2. The system has three states, corresponding to the number of enqueued processes. The first process in the queue represents

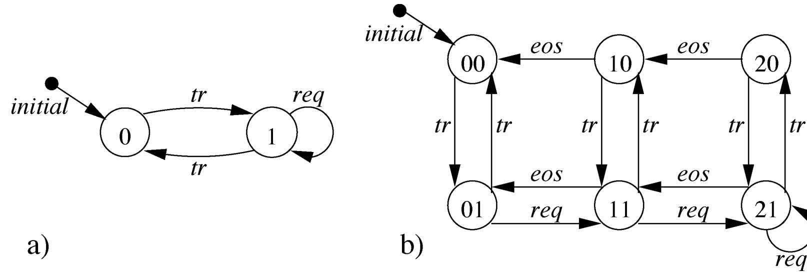


Fig. 2. (a) State transition graph of the two-state client of Example 2. (b) State transition graph of a system composed of the client of Example 2 and the server of Example 1.

the process being served. Triggering events are incoming requests (req) and end of services (eos). Incoming requests are external events, while ends of services are internal events. (See the equation at the bottom of the page.)

The residual service time can be computed based on x_n. For instance, assuming that the service time is one for all requests, the residual service time can be expressed as $r_s = 1 - t_{\text{elapsed}}$, where the elapsed time t_{elapsed} is computed since the last decision epoch at which either state 0 was exited or a request was serviced (i.e., since the decision epoch at which the server started servicing the current request). The state diagram and a possible trajectory of the system are shown in Fig. 1.

DESs require the number of events occurring in a limited time period to be finite. This property is assumed to be verified by external (input) events, while it has to be guaranteed by next-event function $G(x_n)$ for internal events. A finite number of events are allowed to occur simultaneously. Simultaneous events are treated as subsequent events with infinitesimal distance in time. This execution semantic is self-consistent if and only if system evolution does not depend on the order simultaneous events are processed.

B. Compositional Model

Consider two interacting DESs A and B for which a representation is provided according to Section III-A. We model interaction by making A sensitive to (some of) the internal events of B and vice versa. We use superscripts (A) and (B) to refer to the two systems. We say that A and B *interact* with each other if either $E_E^{(A)} \cap E_I^{(B)} \neq \emptyset$ or $E_E^{(B)} \cap E_I^{(A)} \neq \emptyset$ or both.

Example 2: Consider a nonblocking client generating requests for the server of Example 1. The client has two states: active (1) and inactive (0). State transitions are triggered by an external event (tr). When active, the client issues service requests (represented by an internal event req) with exponentially distributed interarrival times, when inactive it does not. We denote by $\texttt{exp}(\mu)$ an exponentially distributed random variable with mean μ

$$
\begin{aligned}
S &= \{0,1\} \\
E_I &= \{\text{req}\} \\
E_E &= \{\text{tr}\} \\
s_{n+1} = F(x_n, \lambda_{n+1}, e) &= 1 - s_n && \text{if } e = \text{tr;} \\
&= s_n && \text{if } e = \text{req;} \\
&= 0 && \text{if } e = \text{initial;} \\
&= s_n && \text{otherwise.} \\
(e, r_e) = G(x_n) &= (\text{req}, \texttt{exp}(\mu)) && \text{if } s_n = 1; \\
&= (\text{none}, \infty) && \text{if } s_n = 0.
\end{aligned}
$$

The client interacts with the server by means of event req. The state diagram of the client is shown in Fig. 2(a).

The representations of interacting systems can be merged to obtain a global representation of the entire system, with

$$
\begin{aligned}
S &= S^{(A)} \times S^{(B)} \\
E &= E^{(A)} \cup E^{(B)} \\
E_I &= E_I^{(A)} \cup E_I^{(B)} \\
E_E &= E - E_I
\end{aligned}
$$

$$
\begin{aligned}
S &= \{0,1,2\} \\
E_I &= \{\text{eos}\} \\
E_E &= \{\text{req}\} \\
s_{n+1} = F(x_n, \lambda_{n+1}, e) &= s_n + 1 && \text{if } e = \text{req}, s_n < 2; \\
&= s_n - 1 && \text{if } e = \text{eos}, s_n > 0; \\
&= 0 && \text{if } e = \text{initial;} \\
&= s_n && \text{otherwise.} \\
(e, r_e) = G(x_n) &= (\text{eos, residual service time}) && \text{if } s_n > 0; \\
&= (\text{none}, \infty) && \text{if } s_n = 0.
\end{aligned}
$$

Example 3: The system composed of the server of Example 1 (subsystem A) and the client of Example 2 (subsystem B) can be described as

$$S = \{(0,0),(0,1),(1,0),(1,1),(2,0),(2,1)\}$$
$$E_I = \{\text{eos}, \text{req}\}$$
$$E_E = \{\text{tr}\}.$$

The corresponding state transition graph is shown in Fig. 2(b).

Notice that the trajectories of two interacting systems at a given time $\bar{t}$ may be composed of a different number of steps, since a system may not be sensitive to some of the triggering events of the other one. We denote by n the number of steps in the trajectory of the global system, and by n_A and n_B the number of steps in the trajectories of A and B, respectively. In general, $n_A \leq n$ and $n_B \leq n$. The current state of the system is $s_n = (s^{(A)}_{n_A}, s^{(B)}_{n_B})$.

Suppose that event $e_{n+1} = e \in E$ occurs at decision epoch $\lambda_{n+1} = t$, causing a new step to be added to the trajectory of the system: $(s_{n+1}, \lambda_{n+1}, e_{n+1})$. If event $e \in E^{(A)} \cap E^{(B)}$, it causes state transitions in both subsystems at time t, bringing their trajectories to $(s^{(A)}_{n_A+1}, \lambda_{n_A+1}, e_{n_A+1})$ and $(s^{(B)}_{n_B+1}, \lambda_{n_B+1}, e_{n_B+1})$. In this case, the relation between the annotated state of the system and those of the subsystems is the following:

$$s_{n+1} = \left(s^{(A)}_{n_A+1}, s^{(B)}_{n_B+1}\right)$$
$$\lambda_{n+1} = \lambda_{n_A+1} = \lambda_{n_B+1} = t$$
$$e_{n+1} = e_{n_A+1} = e_{n_B+1} = e.$$

Now consider the case of subsystem B being insensitive to event e (i.e., $e \notin E^{(B)}$). In this case, event e causes a state transition in subsystem A, but it does not cause any transition in B. Hence, time t is a decision epoch for A (and for the entire system) but it is not for B. The relation between the annotated state of the system and those of the subsystems becomes

$$s_{n+1} = \left(s^{(A)}_{n_A+1}, s^{(B)}_{n_B}\right)$$
$$\lambda_{n+1} = \lambda_{n_A+1} = t$$
$$e_{n+1} = e_{n_A+1} = e.$$

Example 4: A possible trajectory for our client–server system is shown in Fig. 3, together with the corresponding trajectories of the two subsystems. Notice that the decision epochs of the system are the union of those of all subsystems.

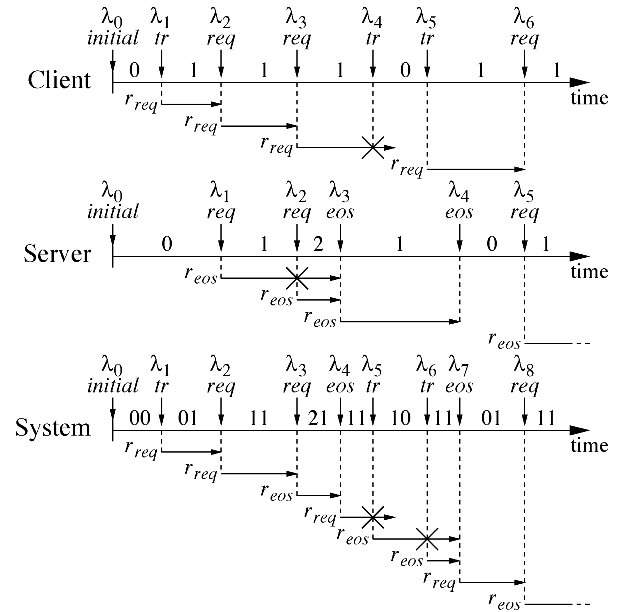


Fig. 3. Trajectory showing the interaction between a two-state client and a three-state server.

For instance, decision epoch λ_4 in the system trajectory corresponds to decision epoch λ_3 in the server trajectory, while it does not appear in the trajectory of the client.

The next-state function of the system (F) can be obtained from those of the subsystems in the following way:

$$s_{n+1} = F(x_n, \lambda_{n+1}, e_{n+1}) = \left(s^{(A)}, s^{(B)}\right)$$

where

$$\begin{aligned} s^{(A)} &= s^{(A)}_{n_A+1} = F^{(A)}\left(x^{(A)}_{n_A}, \lambda_{n+1}, e_{n+1}\right) && \text{if } e_{n+1} \in E^{(A)}; \\ &= s^{(A)}_{n_A} && \text{if } e_{n+1} \notin E^{(A)}. \\ s^{(B)} &= s^{(B)}_{n_B+1} = F^{(B)}\left(x^{(B)}_{n_B}, \lambda_{n+1}, e_{n+1}\right) && \text{if } e_{n+1} \in E^{(B)}; \\ &= s^{(B)}_{n_B} && \text{if } e_{n+1} \notin E^{(B)}. \end{aligned}$$

Composition of next-event functions is even simpler

$$(e, r_e) = G(x_n) = \text{first}\left\{G^{(A)}\left(x^{(A)}_{n_A}\right), G^{(B)}\left(x^{(B)}_{n_B}\right)\right\}$$

where first{ } selects the event that occurs first, and the next-event function of each subsystem is assumed to be evaluated at the last decision epoch for the subsystem.

Example 5: Next-state function F of the client–server system of Example 3 is represented by the state transition graph of Fig. 2(b). Its next-event function G is specified in the equation at the bottom of the page.

$$\begin{aligned} (e, r_e) = G(x_n) &= \text{first}\{(\text{eos}, \text{residual service time}), (\text{req}, \texttt{exp}(\mu))\} && \text{if } s^{(A)}_{n_A} > 0, s^{(B)}_{n_B} = 1; \\ &= (\text{eos}, \text{residual service time}) && \text{if } s^{(A)}_{n_A} > 0, s^{(B)}_{n_B} = 0; \\ &= (\text{req}, \texttt{exp}(\mu)) && \text{if } s^{(A)}_{n_A} = 0, s^{(B)}_{n_B} = 1; \\ &= (\text{none}, \infty) && \text{if } s^{(A)}_{n_A} = 0, s^{(B)}_{n_B} = 0. \end{aligned}$$

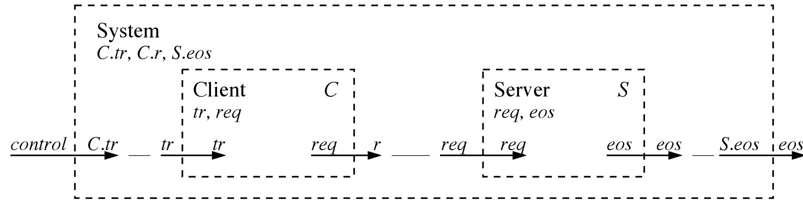


Fig. 4. Example of hierarchical interfaces.

In the following we use the symbol $\odot$ to denote the composition of DESs. A system composed of subsystems A and B will be denoted by $A \odot B$. Composition is an associative and commutative operation: the resulting system does not depend on the order in which subsystems are composed. From a denotational point of view, however, the representation of the physical state of the system as an ordered n-tuple of the physical states of its subsystems depends on the order in which Cartesian products are performed.

The dynamics of a system consisting of interacting subsystems is the result of the composition of the dynamics of its components, each of which evolves based only on its own trajectory, that can be viewed as the projection of the system trajectory on the state and event sets of the subsystem. Subsystems have no visibility of the internal states of other subsystems, so that their evolution is a local (rather than global) phenomenon. Subsystems interact with each other only by means of some shared events.

On the other hand, the evolution of a generic DES may be based on its entire trajectory. If this is the case, the system cannot be obtained by means of composition. A system that can be expressed as the composition of interacting subsystems is called a *decomposable* system. DESs are not always decomposable.

1) Abstraction: In the previous section we said that two DESs A and B interact if there is at least an internal event (say, e) of a system (say, A) that is an external event for the other one (B). In symbols, $e \in E_I^{(A)} \cap E_E^{(B)}$.

Even if intersection of event sets is the key mechanism for interaction, it is impractical, since it imposes consistent naming conventions to be used when defining the event sets of each system: A and B communicate through event e if and only if it appears with the same name in $E_I^{(A)}$ and $E_E^{(B)}$. If different names were used when defining the event sets of A and B, no interaction would be allowed. On the contrary, if the same name was carelessly assigned to different events in A and B, their interaction is enforced. In practice, in order to use intersection for modeling the interaction between DESs, consistent names should be assigned with all the event sets of the components of a complex system, thus avoiding the reuse of a component specification within different systems.

To overcome the above-mentioned drawbacks, we need to make composition independent of the names used within the specification of each subsystem. To this purpose, we associate an interface with each DES that maps local event names onto input and output ports to be used for compositional purposes. The specification of a system, together with its interface, is treated as a macro that can be repeatedly instantiated as a system component. When designing a system, each component has to be assigned with a unique name. Component ports are uniquely identified within a system by using the name of the instance as a prefix for the name of the port. Interaction between two components is explicitly specified by connecting their input–output ports.

Example 6: Interfaces are used in Fig. 4 to represent the client–server interaction of our example system. Interfaces are represented as dashed boxes. The name of the macro is reported on the top-left corner, together with the list of local events. The name of the instance is reported on the top-right corner. Input and output ports are denoted by incoming and outgoing arrows. External and internal labels associated with each port represent the mapping between port names and local event names. Internal event req generated by the client C is made available at output port r. External event req is taken by the server S from input port req. Client–server interaction is represented by the connection between output port $C.r$ and input port S.req

$$C.r \longrightarrow S.\text{req}.$$

Interfaces provide a mechanism for abstraction, in that they hide system specification focusing only on compositional properties. Moreover, interfaces hide all internal events that are not mapped to output ports.

Example 7: In Fig. 4, an interface is also associated with the entire system, viewed as a macro called *System*. Local events for the entire system are the events of its components that are mapped onto output ports and unconnected input ports, namely, C.tr, $C.r$, S.eos. The system interface maps C.tr onto primary input port control and S.eos onto primary output port eos, while it hides internal event $C.r$.

To provide additional flexibility to the compositional model, advanced port mappings can be introduced that combine and duplicate events. Event *combination* maps a set of events $e_1, \ldots, e_N$ onto a single event e that occurs whenever one of the events in the set occurs. In practice, event combination provides a mechanism for simplifying the representation of equivalent events: a system that reacts to event e reacts to any of the events mapped to e.

Event *duplication* maps a single event e into multiple events $e_1, \ldots, e_N$ that occur whenever event e occurs. Duplication represents fan-out points: an event needs to be duplicated to drive different components. Examples of event combination and duplication are provided in the following example.

Example 8: Fig. 5 shows the hierarchical representation of a system composed of three independent clients issuing service requests for the same server. Leaf components are an instance (S) of the server in Fig. 5, two instances (C2.A and

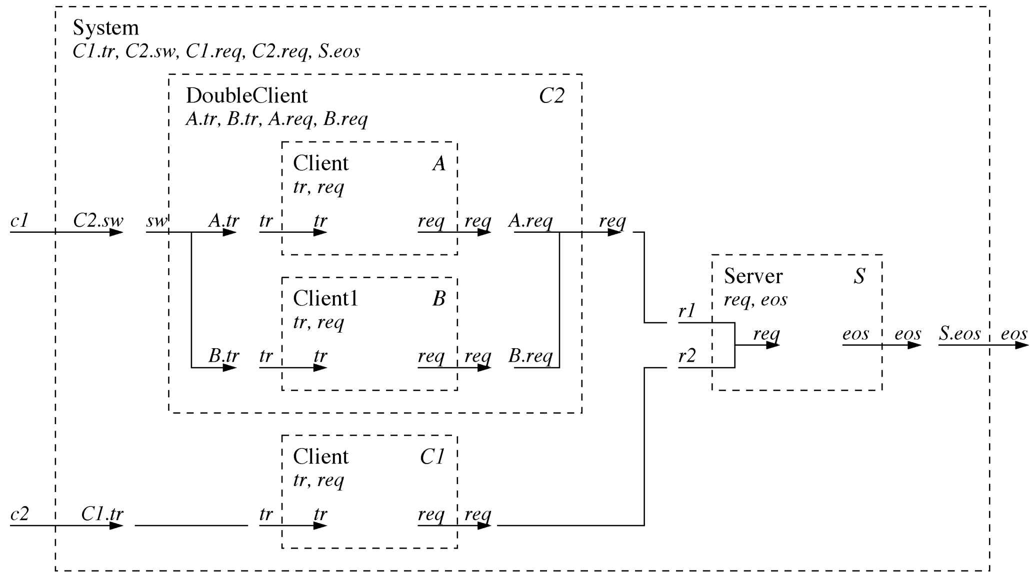


Fig. 5. Example of abstraction.

```
void processEvent(trigEv) {
  nextState = F(x,trigEv.time,trigEv.event);         // chose next state
  x = addStep(x,nextState,trigEv.time,trigEv.event); // update trajectory
  if ((nextEv != NULL) && (nextEv != trigEv))
    cancel(nextEv);                                  // cancel candidate triggering event
  nextEv = G(x);                                     // generate new candidate triggering event
  schedule(nextEv);                                  // schedule candidate triggering event
  n = n+1;                                           // increment decision epoch counter
}
```

Fig. 6. Procedure for the event-driven execution of a generic DES model.

C1) of the client in Fig. 5, and an instance (C2.B) of a different client with initial state 1 (rather than 0). Components C2.A and C2.B are wrapped together to form a new macro (called *DoubleClient*) acting in its turn as a client whose interface exhibits a single input port (sw) and a single output port (req). The interface duplicates input event sw to control both subsystems, and combine their requests to generate output events on req. Since C2.A and C2.B start from opposite initial states and have simultaneous state transitions (triggered by the same external event), they are alternatively active. Hence, *DoubleClient* represents a bimodal workload controlled by an external event. As for the server, it has two input ports, r1 and r2, driven by C1 and C2. However, the two ports are combined by the interface in the same input event, making requests from the two clients indistinguishable for the server.

A global interface is provided for the system, hiding internal events C2.req and C1.req.

C. Executable Model

An executable model for a DES is a model that provides executable specifications for next-state function F and next-event function G. Without loss of generality, we assume that the executable description is to be executed by an event-driven simulator developed in a procedural programming language. System specification is provided by implementing the `processEvent` function that is called by the scheduler whenever a triggering event has to be processed. We also assume that the scheduler supports `schedule` and `cancel` functionalities to allow the system to add to the event queue the internal triggering events it generates and to remove from the queue triggering events overcome by external events. The pseudocode of a general `processEvent` procedure is shown in Fig. 6. Events are treated as structures containing both the event type and the occurrence time. The pseudocode refers to global variables of the system: the trajectory `x`, initialized to NULL, the candidate next triggering event `nextEv`, initialized to NULL, and the counter of decision epochs `n`, initialized to zero.

The triggering event is passed to `processEvent` as input parameter `trigEv`. Next-state function `F` is called to chose the target `nextState` for the current transition and the corresponding step is added to system trajectory `x` (using function `addStep`). If `trigEv` is an external event, then the internal candidate event scheduled at last iteration (stored in variable `nextEv`) needs to be removed from the event queue. This is done by invoking the `cancel` procedure. A new candidate event is then generated by next-event function `G`, stored in variable `nextEv`, and scheduled by calling function `schedule`. Finally, the counter of decision epochs `n` is incremented and execution control returned to the scheduler.

The system description is specialized by defining functions `F` and `G` that implement next-state and next-event functions, respectively.

```
nextState F(x, time, event) {
  nextState = getNextState(x,event,time);           // chose next state
  s = getLastState(x);                              // get current state
  deltaT = time - getEnterTime(x);                  // evaluate elapsed time
  for each event e {                                // update clock readings
    if (e is active in nextState) {
      if ((e.clk == -1) || (e == event)) e.clk = 0;
      else e.clk += deltaT * speed[s][e];
    } else e.clk = -1;
  }
  return(nextState);                                // return next state
}
```

Fig. 7. Implementation of the next-state function of a GSMP.

```
nextEvent G(x) {
  nextTime = INFTY;
  for each active event e
    r = getResTime(x,e);                            // generate residual time for event e
    if (r < nextTime) {
      nextTime = r;                                 // update next time
      nextEventName = e;                            // update next event name
    }
  if (nextTime < INFTY)
    nextEvent = newEvent(nextTime,nextEventName);  // generate nextEvent
  else nextEvent = NULL;
  return(nextEvent);                                // return nextEvent
}
```

Fig. 8. Implementation of the next-event function of a GSMP.

D. GSMPs

GSMPs are DESs with nondeterministic next-state and next-event functions based on conditional next-state probabilities and residual-time distributions, respectively.

Conditional next-state probability $p(s' : x_n, e_{n+1}, \lambda_{n+1})$ represents the probability of entering state s' at decision epoch λ_{n+1}. In general, such probability may depend on the entire history of the system, represented by its current trajectory x_n, by current decision epoch λ_{n+1}, and by the event e_{n+1} that triggered the current state transition. We introduce a random variable s, defined on the physical state set S, and we define a *next-state distribution*

$$\mathcal{F}(s : x_n, e_{n+1}, \lambda_{n+1}) = \text{Prob}\{s' \leq s : x_n, e_{n+1}, \lambda_{n+1}\}.$$

Distributions are more general than probability functions, in that they can be defined for both finite and infinite state spaces. As long as a family of next-state distributions is available, the evaluation of the next-state function reduces to the selection of the distribution associated with the current history and to the generation of a random next-state from such a distribution. This is done by the first row of the pseudocode of Fig. 7, where `getNextState()` returns a pseudorandom value from a distribution possibly dependent on the system trajectory `x`, on the triggering event `event`, and on the current time `time`.

Residual-time distributions are associated with each internal event. Whenever a physical state is (re)entered, residual times are generated for all internal events according to their distributions. The internal event with the smallest residual time is then chosen as candidate triggering event. Residual-time distributions may be conditioned to the history of the system, that is entirely represented by its trajectory up to the decision epoch at which current state has been entered. We denote by r_e the residual time of event e and by $\mathcal{G}_e$ its distribution

$$\mathcal{G}_e(t : x_n) = Prob\{r_e \leq t : x_n\}. \quad (1)$$

The general pseudocode of a next-event function based on residual-time distributions is shown in Fig. 8. Residual times are generated for all active events by calling the `getResTime` function. The event with the lowest residual time is then generated and returned to be scheduled as `nextEvent`.

Any system obtained by composition of GSMPs is a GSMP.

1) Clock Structure: Equation (1) provides the flexibility required to specify arbitrary GSMPs, since it allows residual-time distributions to depend on the trajectory of the system, that represents its entire history. However, expressing residual times as stochastic functions of x_n is often unnatural. In most cases, simpler data structures, carrying only partial information about the past, provide a more natural support for the specification of residual-time distributions. Such a natural support is provided by *clocks*, introduced by Glynn as part of the GSMP formalism [27].

A clock c_e can be associated with event e to measure the amount of time elapsed since a past decision epoch chosen as

a (temporary) reference for that event. At any decision epoch, the clock is either restarted (thus updating the time reference for the event) or incremented of a quantity proportional to the last state-holding time. In practice, the value of c_e (called *clock reading* and denoted by the same symbol used for the corresponding clock, c_e) provides a partial view of the history of the system tailored on event e.

Clocks are associated with all internal events. For each state $s \in S$, a set $E_I(s) \subset E_I$ of internal *active events* is defined. Active events for state s are those internal events that may trigger transitions from s. A *clock speed* $v(s,e)$ is associated with each pair (s,e), with $s \in S$ and $e \in E_I(S)$. Clock speed $v(s,e)$ is used to scale the time spent in state s when updating c_e. Clock c_e is restarted either when event e becomes active (i.e., when a state s' is entered from state s, with $e \in E_I(s')$ and $e \notin E_I(s)$) or when event e occurs (i.e., when event e is the triggering event). Hence, c_e represents the time elapsed from the last occurrence/activation of event e. When event e is inactive, clock reading c_e is undefined.

Figs. 7 and 8 show the pseudocode of next-state and next-event functions of general GSMPs. Clock readings and speeds are treated as fields of the data structures used to represent events. When the event is inactive, the corresponding clock is set to conventional value -1. Since clock readings are updated whenever a state is exited, this is done by `F()`, which implements the next-state function. Notice that clock readings do not appear explicitly in function `G()` of Fig. 8, since they are passed to the `genResTime` function as fields of data structure e.

It is worth noting that clocks are not necessary. All clock readings at decision epoch λ_n could be computed from trajectory x_n and clock speeds $v(s,e)$. In practice, clocks are nothing but event-specific partial views of the history of the system that can be used to simplify the specification of residual-time distributions. In most cases of practical interest, in particular, current clock readings and present state s_n retain all the information about the past that may affect the dynamic of the system, thus completely replacing trajectory x_n. In this case (1) can be rewritten as

$$\mathcal{G}_e(t : s_n, c_e) = Prob\{r_e \leq t : s_n, c_e\} \tag{2}$$

and the pseudoalgorithms of Figs. 7 and 8 can be simplified accordingly.

In the following, we implicitly refer to GSMPs whose memory of the past is completely represented by clock readings and current state. We will show in the application section that this assumption does not limit the modeling power of GSMPs, allowing us to model with no approximation real-world systems with arbitrary event distributions.

2) Dealing With Infinite States and Infinite Events: The implementation of next-state and next-event functions described so far implicitly assume that both the event set and the state set are finite. If this not the case, infinite conditional distribution functions should be specified in order to fully describe the GSMP, and next-state and next-event functions would never exit their inner loops.

On the other hand, many real-world systems do have infinite states and infinite events. Consider, for instance, a digital system whose clock frequency can be dynamically controlled by an external command in a continuous interval from $f_{\min}$ to $f_{\max}$. In this case, infinite input events are required to represent external commands, and infinite states are required to represent the operation state of the system. However, all external events representing dynamic control of the clock frequency differ only for a parameter that is the target clock frequency. Similarly, all active states of the system differ only for their operation frequency. This suggests that the entire set of events (states) could be represented as a single event (state) associated with a continuous parameter.

In general, we are interested in GSMPs with a finite set of (possibly parameterized) states, a finite set of (possibly parameterized) external events, and a finite set of (possibly parameterized) internal events. Parameterization provides an implicit finite representation of both finite and infinite (continuous or discrete) state/event sets, depending on the nature of the parameters.

We call *state class* (*event class*) any subset of states (events) represented by a unique parameterized state (event).

The actual state of the system is represented by its current state class and by the current configuration of all parameters associated with it. Similarly, any triggering event is represented by the event class it belongs to and by a unique configuration of the parameters associated with it. We also assume a unique clock to be associated to an entire event class, meaning that all events belonging to the same class share the same clock readings and that the clock is reset upon the occurrence of any event of the class.

The pseudocodes of next-state and next-event functions reported in Figs. 7 and 8 can be simply extended to handle parameterized states and events by representing parameters as fields of states and events structures. In particular, function `getNextState` that appears in Fig. 7 will return a random value according to a distribution depending not only on the current state and triggering event, but also on their parameters. In other words, the next-state distribution may be parameterized as well. Moreover, the returned value will be not only a value, but a data structure representing the destination state class together with the configuration of all parameters possibly associated with it.

As for the next event, function `getResTime` returns for each event class a residual time from a distribution that may depend on the configuration of the parameters associated with the current state. However, the residual-time distribution is unique for all events of class `e`. This means that the residual time returned by the function represents the first occurrence of any event of the class. The actual triggering event (i.e., the configuration of the parameters associated with the event class with the minimum residual time) will be selected by function `newEvent`, according to a given event distribution that is a property of the event class and that may depend on the residual time.

This implementation is consistent to the GSMP semantics, since both the residual-time distribution and the event distri-

bution for a given event class could in principle be obtained from the residual-time distributions of each event in the class

$$\mathcal{G}_e(t : s_n, c_e) = Prob\{r \leq t : s_n, c_e\} = 1 - \prod_{\epsilon \in event\ class\ e} (1 - \mathcal{G}_\epsilon(t : s_n, c_e)) \quad (3)$$

$$\mathcal{E}_e(e : s_n, t) = Prob\{\epsilon \leq e : s_n, t\}. \quad (4)$$

However, when dealing with infinite state sets, it is more practical and intuitive to directly specify $\mathcal{G}$ and $\mathcal{E}$ for the entire class.

We remark that in many cases, parametric states/events may be useful not only to provide a finite representation of infinite sets, but also to provide a compact representation of finite sets. This is the case, for instance, of a finite queue of length N, whose N states could be represented by a unique state class with an integer parameter ranging from zero to N.

E. System Taxonomy

The GSMP formalism introduced in the previous section allows us to describe very general systems. Since our goal is to model realistic DPM systems, we can make several simplifying assumptions to restrict the modeling space. Some of these assumptions were already mentioned and discussed in the previous sections, but we list them here for the sake of completeness. Namely, we assume GSMP models with:

1) a finite set of state classes;
2) a finite set of internal event classes;
3) a finite set of external event classes;
4) next-state distributions depending only on the current state and on the triggering event;
5) clocks associated with each event class;
6) residual-time distributions defined for each event class, depending only on the current clock readings and on the current state;
7) event distributions defined for each event class as functions of the current state and of the triggering time.

In the following, we provide a classification of DESs directly induced by the proposed modeling framework. With respect to the state structure, a system is said to be:

- *continuous state* if there is at least a continuous parameter associated with one of its state classes;
- *discrete state* if there are no continuous parameters associated with state classes;
- *finite state* if none of the parameters associated with state classes may take an infinite set of values.

A similar classification is induced by the event set. A system is:

- *continuous event* if there is at least a continuous parameter associated with one of its event classes;
- *discrete event* if there are no continuous parameters associated with event classes;
- *finite event*, if none of the parameters associated with event classes may take an infinite set of values.

Depending on the number of internal events made observable from the external interface, a system/component may be *unobservable*, *partially observable*, or *totally observable*. Also, the system can be either *controllable* or *autonomous* depending on its sensitivity to external events.

A system composed of multiple interacting GSMPs is said to be *decomposable*, while it is *monolithic* otherwise. Notice that subsystems communicate only by means of events: the current state of a subsystem is not visible to other subsystems unless specific events are used to notify state changes. Hence, observability and controllability are key properties for intercomponent communication. The interactions between the subsystems of a decomposable GSMP can be represented by means of a dependency graph with nodes associated with GSMPs and directional arcs associated with event-passing communication. The dependency graph has an arc from component A to component B if an only if A is observable from B and B is controlled by A. Autonomous and unobservable components are represented as source and sink nodes in the dependency graph.

In general, GSMPs may be either *stationary* or *nonstationary* depending on whether conditional distributions depend on time or not. We are mainly interested in modeling stationary systems. As for the timing model, discrete events may occur at any point in time, provided that a finite number of events occur in a limited period. Hence, in general, events may be associated with continuous residual-time distributions. Nevertheless, slotted-time systems can be modeled by means of discrete residual-time distributions. A system has a *slotted-time* model if and only if all external and internal events are associated with discrete residual-time distributions with a common discretization step.

GSMP systems are inherently nondeterministic, since both next-event and next-state functions are specified by means of conditional probability distributions. However, deterministic decisions can be taken according to deterministic distributions, returning always the same value. A system is said to be *deterministic* if its model has only deterministic next-state and residual-time distributions; it is said to be *nondeterministic* if its model has at least a nondeterministic distribution. A deterministic GSMP may have nondeterministic input events, but it cannot generate nondeterministic internal events.

A nondeterministic GSMP model is a *Markov* model if and only if all residual times have exponential distributions depending only on the present state.

Example 9: A variable-frequency digital system has a continuous state space if the clock frequency can take any value in a given range, while it has a finite state space if it can work at a finite number of clock frequencies. In both cases, the system is controllable if frequency adjustments are triggered by external events. The input events are either continuous or finite, according to the state space of the system. An unlimited first-in, first-out (FIFO) queue has an infinite discrete state set and a finite event set. A nonblocking client can be modeled as an autonomous system (since it is not sensitive to any external event). The simple server of Example 1 is completely observable, since it has a unique internal event (eos) that is made observable through an output port. A server with constant service time is deterministic, while a server with exponential service time is a Markov process.

F. DPM

DPM entails the interaction between (at least) a PMS and a PM. According to our taxonomy, any PMS has a controllable model, since it reacts to external events representing DPM commands issued by the PM. On the other hand, the PMS is usually (at least partially) observable from the PM, so that their interaction can be represented by a cyclic dependency graph. In most cases, the system is also controlled by a workload that can be modeled as an additional component. The model of the workload can be either autonomous or controlled in its turn by the system. For instance, a nonblocking client can be modeled as an autonomous GSMP, while a blocking client is controlled by the end-of-service events issued by the system. The workload may or may not be observable from the PM.

1) Design Metrics: The PM implements a control policy aimed at optimizing a system metric (e.g., average power consumption, energy per task, etc.) while meeting the constraints possibly imposed to other metrics (e.g., performance, quality of service, etc.). We call *metric* any function of the system trajectory that does not affect the evolution of the system. Hence, metrics are not conceived to be visible to system components. Rather, they can be used by the designer to evaluate/optimize the control policy implemented by the PM. According to an event-driven execution paradigm, metrics are evaluated (i.e., updated) only at decision epochs.

Any function of system metrics is also a system metric. Our definition of metrics is very general. Hence, we can use our framework to deal with system objectives and constraints of different types, including, but not limited to, average values, worst case values, variances, etc. The expressive power and flexibility of our approach stems from two specific factors. First, the simulation-based approach can incorporate arbitrary functions (for instance, we are not constrained to linearity as in the case of linear-programming optimization of Markov decision processes [53]). Second, the evaluation of the metrics does not affect system trajectory, so that metrics are not subject to constraints possibly imposed by the modeling framework.

2) Optimum Control: The degrees of freedom exploitable for DPM are represented by the external events the PMS is sensitive to. Similarly, all information available for taking DPM decisions are provided by the observable internal events of all system components, possibly including the workload.

We call *ideal controller* a PM that fully exploits the DPM potential of the system by: 1) taking trace of all observable events; 2) controlling all controllable events; 3) implementing arbitrarily complex DPM policies; and 4) using unlimited computational and memory resources. We call *oracle* an ideal controller that has complete knowledge of past and future observable events.

For our purposes, both ideal controllers and oracles represent unrealistic components to be used only for determining bounds on achievable metrics. However, while ideal controllers may be modeled and simulated within the DES/GSMP framework, oracles cannot, since they are noncausal systems. Nevertheless, in many practical situations, it is possible to evaluate the effect of an oracle PM by analyzing offline full traces of events.

Any *real controller* is a causal PM that has reduced capabilities as compared to the ideal one because of resource limitations that may impose partial observation of the observable events, partial control of the controllable events, approximated representation of past history, simple probabilistic structures, and limited decision epochs.

The nature of a PM is fully determined by its state structure and event set, while its behavior is determined by next-state and next-event functions, to be specified in terms of next-state and residual-time distributions. In particular, the behavior of the controller has to be optimized in order to minimize/maximize a given function of system metrics (called *objective function*) while meeting constraints imposed on some other metrics.

The results of optimization are next-state and residual-time distributions that specify the *policy* implemented by the controller. Policy optimization is said to be *parametric* if the probability distributions have a fixed form and tunable parameters. For instance, a Markovian PM has only exponential residual-time distributions, so that the expected values of the residual times are the only degrees of freedom available for optimization. Optimization is called *nonparametric* if the distributions do not have a fixed form.

Policy optimization is a complex task that in most practical cases need to be addressed by means of heuristic algorithms leading to suboptimal solutions.

Given a PMS and a workload, assume that optimum policies have been found for an ideal controller, for an oracle and a for a real controller, according to a given constrained optimization problem aiming at minimizing an objective function f. We denote by f_{ideal}, f_{oracle}, and f_{real} the values of the objective function achieved when using as PM the ideal controller, the oracle and the real controller, respectively, implementing their optimum policies. The following relation holds:

$$f_{\mathrm{oracle}} \leq f_{\mathrm{ideal}} \leq f_{\mathrm{real}}.$$

Notice that both f_{oracle} and f_{ideal} can be viewed as inherent properties of the PMS and of the workload. In particular, f_{ideal} represents the lower bound of the objective function achievable by any real PM, while f_{oracle} represents the value of the objective function that could be achieved by making always correct predictions about future events. The difference between f_{ideal} and f_{oracle} is a measure of the effect of nondeterminism, while the difference between f_{real} and f_{ideal} is a measure of the nonideality of the real PM.

In some cases, the implementation of the best policy of an ideal controller may require finite memory and computational resources. In this case, a real PM can act as an ideal controller, leading to $f_{\mathrm{real}} = f_{\mathrm{ideal}}$. This is the case, for instance, of a Markovian PMS with a Markovian workload, for which the policy optimization problem can be exactly formulated and solved by means of linear programming [53] and

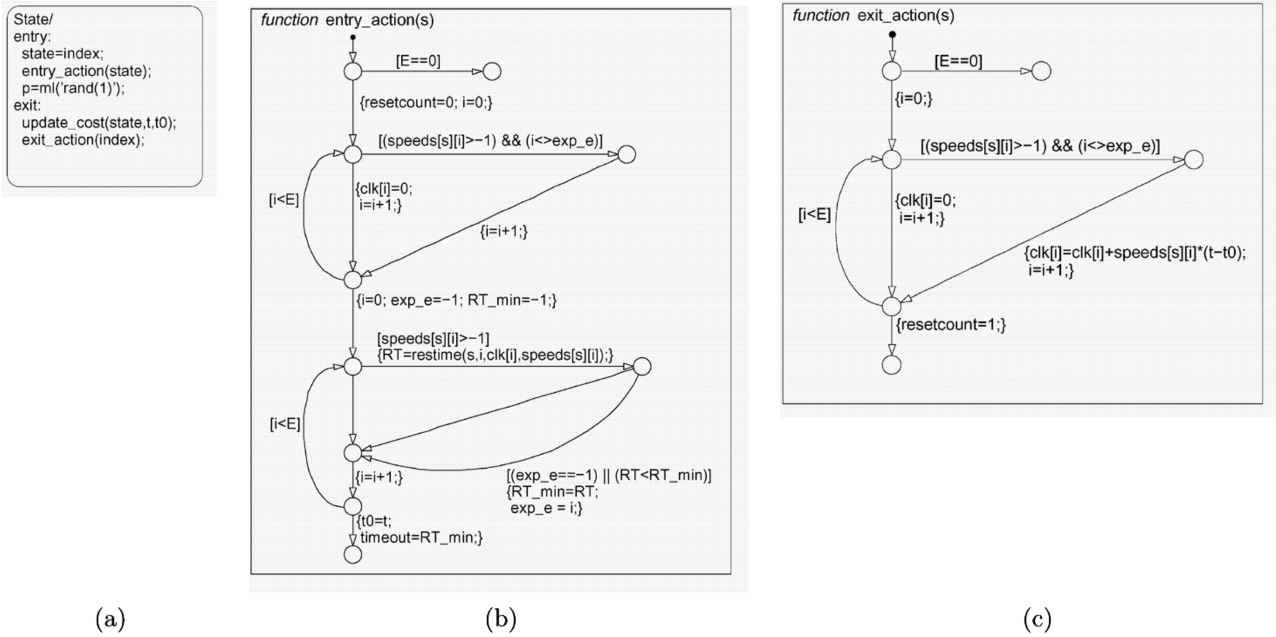


Fig. 9. Template of the state of a GSM component. (a) State structure. (b) Flowchart of the `state_entry` action. (c) Flowchart of the `state_exit` action.

the optimum policy can be implemented by a real PM. In this case, the measure of nonideality is $f_{\text{real}} - f_{\text{ideal}} = 0$.

Furthermore, when both the model of the PMS and the model of the workload are deterministic, future events are always predictable, so that $f_{\text{ideal}} = f_{\text{oracle}}$. In fact, for a deterministic system the measured effect of nondeterminism is null: $f_{\text{real}} - f_{\text{ideal}} = 0$.

IV. GSMP Simulation

Several tools have been developed in the last decade for the specification, analysis, and simulation of GSMP models, mainly focusing on formal verification, performance evaluation, and dependability analysis [52], [54]. In particular, a tool for compositional GSMP modeling and simulation called *GMSim* was proposed by Nilsen [52].

In this section we present a GSMP infrastructure built on top of MathWorks' Simulink [28] for the specification and simulation of DPM systems described as interacting DES/GSMPs. Simulink supports interactive modeling, simulation, and analysis of multidomain dynamic systems specified by instantiating and interconnecting standard and custom library blocks. Simulink modules may interact seamlessly with any Matlab toolkit, providing immediate access to a large set of analysis and design tools.

By using Simulink as a simulation platform we trade off some performance (with respect to dedicated C/C++ implementations) for usability and flexibility. In particular, we leverage the graphical user interface (GUI) of Simulink, its module libraries and its interface to Matlab. The GSMP models implemented in Simulink can interact with other Simulink components, including continuous-time dynamic systems. This provides the flexibility for modeling real-word DPM systems at different levels of abstraction, while retaining the key properties of GSMPs.

In the following, we describe the implementation of the state and timing structure of a GSMP and the interaction between multiple GSMPs.

A. Template of a Basic GSMP Component

The Stateflow toolkit [55] provides the ideal support for the specification of the state structure of a GSMP component. A stateflow component is specified in terms of states, state transitions, and triggering events. In addition, attributes (i.e., data structures) can be associated with the stateflow and with each state, and actions can be associated with events and state transitions.

The global attributes of the stateflow model of any GSMP component include the number of events (`E`), the number of states (`S`), an array of E clock readings (`clk[e]`), a matrix of SxE clock speeds (`speeds[s][e]`), a random variable (`p`), a state-entry time (`t0`), a current state variable (`state`), an expected event index (`exp_e`), and a residual time (`timeout`). Each state has additional local attributes, such as the state identifier (`index`), the power consumption level (`power`), and any user-defined state-dependent metrics.

The evolution of a GSMP is fully determined by the current state, by the clock readings of active events, and by the triggering event. Whenever a new state is entered, the active event set is updated, residual times are generated for all active events based on their clock readings and speeds, and the event with the lower residual time is chosen and scheduled as the next triggering event. When a state is exited, the clock readings of all active events are updated based on their speeds and on the time spent in that state. The support for the implementation of both tasks is provided by the entry and exit actions that can be associated with each state. The template of a state of a GSMP is shown in Fig. 9(a). It consists of three elements: the name of the state (`State`), the state-entry action

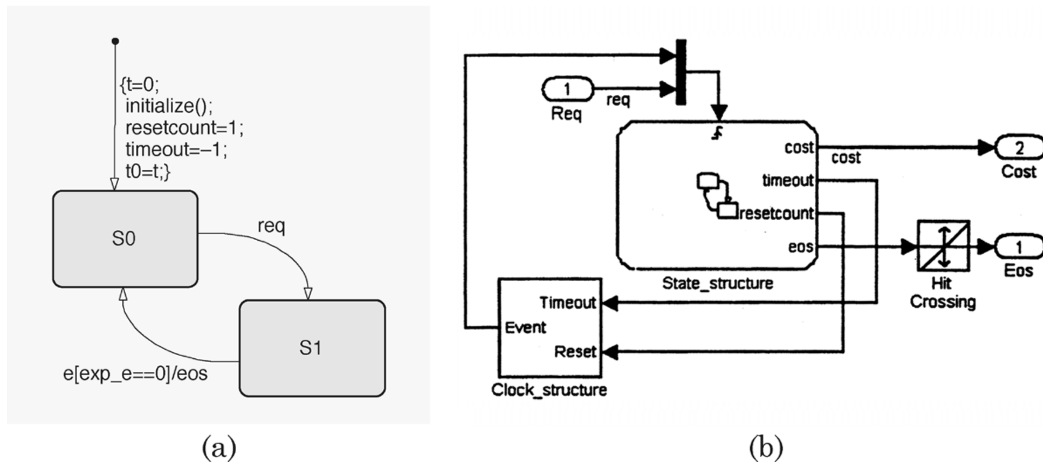


Fig. 10. Simulink representation of a two-state server. (a) Stateflow view. (b) Overall component with interface ports.

(specified under keyword `entry`:), and the state-exit action (specified under keyword `exit`:).

The state-entry action consists of three steps: the assignment of the value of the local variable `index` (that is a unique state identifier) to the global variable `state` (that represents the current state), the invocation of the `entry_action` function (described below), and the generation of a random variable `p` uniformly distributed between zero and one (possibly used to make a nondeterministic choice among outgoing edges triggered by the same conditions). The flowchart of the `entry_action` function is shown in Fig. 9(b). Edge conditions are written in square brackets, while actions are written in curly brackets. The `entry_action` function consists of two loops that scan the entire event set: the first loop resets the clock readings of the last event (i.e., the event that triggered the transition to the current state) and of the events that are inactive in the current state; the second loop generates residual times for all active events (according to their state-dependent distributions) and selects the next triggering event by assigning its index to variable `exp_e` and its residual time to variable `timeout`. Finally, the state-entry time is assigned to variable `t0`. Residual times are generated by function `restime` that takes as input the current state `s`, the event index `i`, the reading of the clock associated with event `i`, and its speed while in `s`. The residual-time distribution to be used for each (state,event) pair is specified within the `restime` function by invoking specific Matlab m-functions belonging to a predefined library of distributions.

The state-exit action is specified by means of two functions: the `update_cost` function that updates global metrics according to the values associated with the current state and to the time spent it that state, and the `exit_action` function [shown in Fig. 9(c)] that updates the clock readings of all active events according to their local speeds and to the time spent in the current state $(\mathtt{t} - \mathtt{t0})$.

Fig. 10 shows the stateflow of the two-state server of Example 3. States S0 and S1 are derived from the template of Fig. 9 and assigned to unique names, unique identifiers, and different costs representing power levels (e.g., cost 0.5 to state S0, representing a waiting condition; cost 1 to state S1, representing a busy condition). Transitions from S0 to S1 are triggered by input event `req`, which represents an incoming service request, while transitions from S1 to S0 are triggered by an internal event that represents the end of service. Since the end of service is the only internal event in this simple example, its index is zero. The triggering condition corresponding to the occurrence of internal event 0, reported on the edge from S1 to S0, is $\mathtt{e}[\mathtt{exp_e} == 0]$. In fact, since there is a single internal event scheduled at each time, all internal events are implemented by a unique simulation event `e`, and a global variable (`exp_e`) (updated whenever a new event is scheduled) is used to recognize the incoming event.

Notice that although event `e` is used to represent internal events, it is not generated within the stateflow model. Rather, it is generated by an external dynamic component that implements the actual clock structure and taken in input by the stateflow. This is shown in Fig. 10(b), which provides a schematic representation of the template of a generic GSMP model. The `State_structure` component is a stateflow module that takes as input one or more triggering events and generates a timeout value corresponding to the residual time of the last scheduled internal event. The `Clock_structure` is implemented as a timeout: it takes the `timeout` value provided by the stateflow and generates event `e` when the timeout has elapsed. Fig. 10(b) also shows the input and output ports belonging to the interface of the GSMP component: input port `Req` provides the input event `req`, output port `Eos` makes the internal event `e` available for interaction with other components, output port `Cost` provides runtime information about a generic cost metric (e.g., power consumption).

Notice that input events are provided to the stateflow component through a unique triggering port. A multiplexer is used to this purpose, as shown in Fig. 10(b).

Internal events are made observable from outside the component by means of additional output events, specified within the stateflow module. To associate an output event with the corresponding internal event, the output event has to be issued by the stateflow model whenever the internal

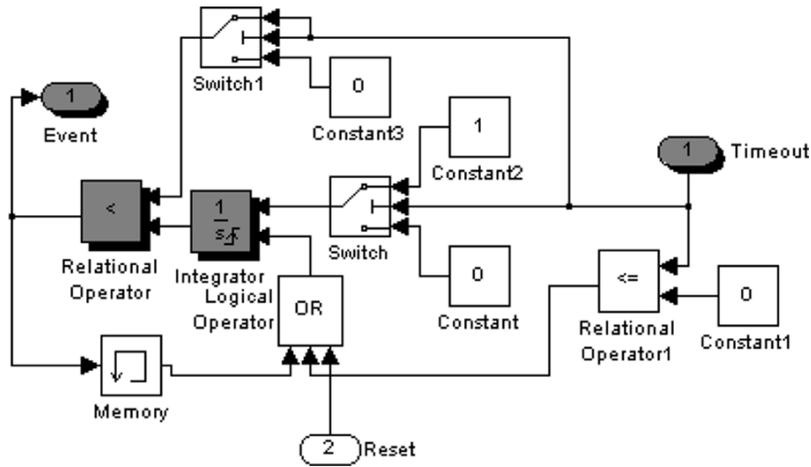


Fig. 11. Implementation of the local clock structure of a GSMP component. The key elements are shaded in the schematic. A timeout value is taken in input and compared to the output of an integrator that generates a linear ramp representing the elapsed time. As the elapsed time passes the timeout values, an output event is generated.

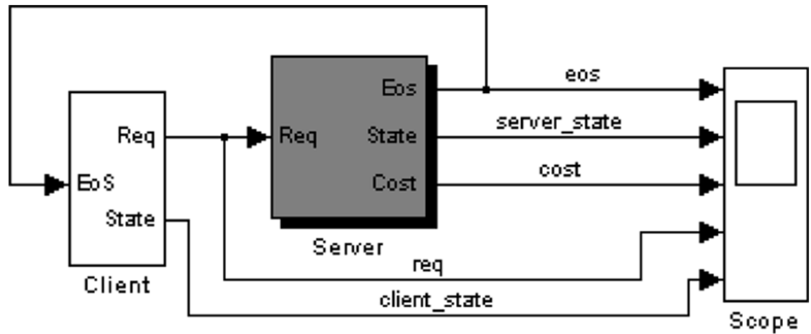


Fig. 12. Schematic representation of a client–server system. An additional output port (`State`) has been added to each component to monitor the internal state for verification and debugging. A standard Simulink component (`Scope`) is used to capture and display all signal waveforms.

event occurs. In particular, the stateflow generates a given event whenever a given state transition is traversed, if the name of the event is appended (after a slash) to the label of the state transition. For instance, the state diagram of Fig. 10(a) specifies that output event `eos` has to be issued whenever there is a transition from S1 to S0, i.e., whenever internal event 0 occurs. The output event `eos` is connected to the output port `Eos` of the GSMP component through a `Hit Crossing` module (provided with the Simulink library) that has the only purpose of making the representation of the output event compatible with the representation of the input events. This is shown in Fig. 10(b).

The inner implementation of the timeout component implementing the clock structure of the GSMP is shown in Fig. 11. It consists of an integrator that generates a linear ramp representing the time elapsed from last reset. The ramp is compared with the input timeout and an event is generated whenever the elapsed time exceeds the timeout. The control circuitry resets the integrator whenever either the output event is triggered or an input reset signal is received.

The interaction among multiple GSMP components is simply obtained by connecting their input/output event ports as shown in Fig. 12 for a client–server example. Additional output signals can be used to observe the system behavior and to evaluate cost/performance metrics. All signals can be monitored by means of standard Simulink components, such as the `Scope` used in the example of Fig. 12.

B. Nondeterministic Destination States

In some cases, the same event may trigger several mutually exclusive transitions from a given state. Each transition may lead to a different destination. Whenever the triggering event occurs, the actual outgoing transition from the current state is randomly chosen among those enabled by the same event, according to a given distribution.

In our implementation, a random number p, uniformly distributed between zero and one, is generated and stored as a global variable whenever a state is (re)entered. The random number can then be used to chose among alternative transitions triggered by the same event. This is done by specifying disjoint conditions (based on p) on each edge.

Example 10: A simple example of a GSMP component with nondeterministic state transitions is shown in Fig. 13. It represents a bursty client that generates bursts of requests with two different distributions when in states `Burst1` and `Burst2`. Transitions from `Idle` to `Burst1` and `Burst2` are triggered by the same event and randomly chosen according to complementary conditions (specified in square

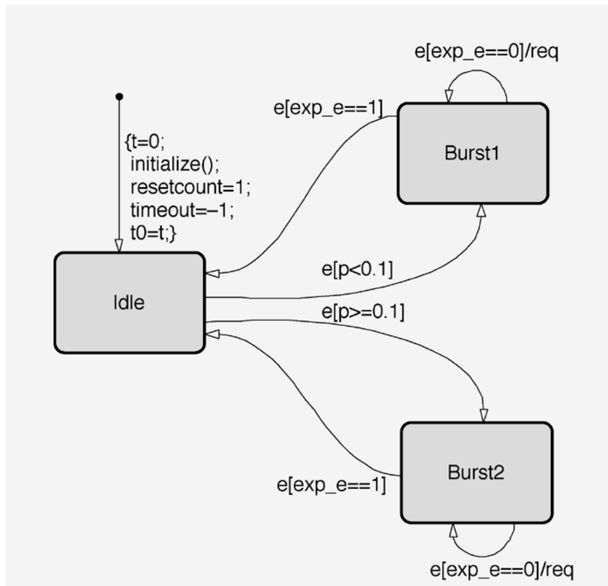


Fig. 13. State structure of a bursty client specified using nondeterministic state transitions.

brackets) based on the observation of random variable p: if p < 0.1 (10% probability) state `Burst1` is entered, if p ≥ 0.1 (90% probability) state `Burst2` is entered. Notice that transitions from `Idle` are not conditioned on the value of `exp_e` (which represents the triggering event), since there is only one event active in state `Idle`. On the contrary, the outgoing edges from states `Burst1` and `Burst2` are conditioned to the value of `exp_e`, since there are two active events (0 and 1) that trigger different transitions.

C. Parameterized States and Events

The parameterized state classes introduced in Section IV-C may be implemented in Simulink as stateflow states with additional parameters whose configuration is used to distinguish among the states belonging to the same class. For instance, a FIFO queue of length N has $N+1$ states that can be modeled as a unique state class with an integer parameter $n \in \{0, \ldots, N\}$ that represents the current number of elements in the queue. Arrivals and departures are external events (`put` and `get`) that trigger transitions among the states of the class. Such transitions are simply implemented as self-loops causing parameter n to be incremented or decremented.

Notice that when $n = 0$ ($n = 1$), the queue is empty (full) and n cannot be further decremented (incremented). These boundary conditions can be implicitly modeled by conditioning the incrementing/decrementing self-loops to the current value of parameter n, as illustrated in Fig. 14(a). An external event `ack` can be generated whenever a self-loop is traversed, in order to notify to the user the execution of `put` and `get` commands.

Similarly, stateflow states can be associated with floating-point parameters in order to represent infinite state classes.

The implementation of a parameterized event is not straightforward, since in our stateflow model events do not carry data values. Hence, a parameter associated with an event is an additional property (i.e., data structure) to be added to any stateflow component that handles the event. Consider, for instance, a producer that puts in the queue either one or two elements at the time. We could use two different events (e.g., `put1` and `put2`) to represent the production of single and double elements, or we could use a single event (`put`) with an integer parameter $m \in \{1, 2\}$. In this latter case, parameter m is a variable whose value has first to be set before issuing event `put`, and then taken into account when processing the event.

For instance, if the parameterized command controls the FIFO queue of Fig. 14, the value of parameter m is observed whenever event `put` is received and it affects both the triggering conditions and the update of the internal state of the queue. This is shown in Fig. 14(b).

Representing different events by means of a single parameterized event enhances model simplicity and scalability. For instance, the model of Fig. 14(b) does not require any modification to be extended to handle arbitrary numbers of simultaneous incoming elements.

It is worth noting that whenever a parameterized event goes across the interface of a stateflow component, separate ports are required for the event and for its parameters.

D. Specification

The specification of a DES/GSMP executable model in Simulink entails the instantiation and specialization of a template module. If the system is decomposable, a Simulink model has to be provided for each component.

The template of a GSMP component is shown in Fig. 15(a) and 15(b). It includes the clock structure that generates internal events and a state structure with a unique reset state (`S0`), derived from the template of Fig. 9, with an incoming edge representing the default initial transition. The stateflow model also includes the specification of functions `initialize`, `entry_action`, and `exit_action`, and the data structures needed to represent internal events, clock readings, and speeds. By default, a unique internal event is defined (event 0) and its speed is initialized to -1. A basic library of residual-time distributions is also provided. In practice, the template is a working executable model of a single-state GSMP with no input, no output, and a single internal event that is never active.

The procedure to create a new GSMP model starting from the template is outlined below step by step.

1) *Creation*. Copy the template component and assign it a new name.
2) *Allocation of data structures*. Set the number of internal events `E` and states `S` and change the size of the data structure used to represent clock readings and speeds accordingly.
3) *Definition of external events*. Add external (input) events by using the `AddEvent` command of the `Explore` tool provided within the Stateflow toolkit [55].
4) *Definition of observable events*. Add observable (output) events as in the previous step.

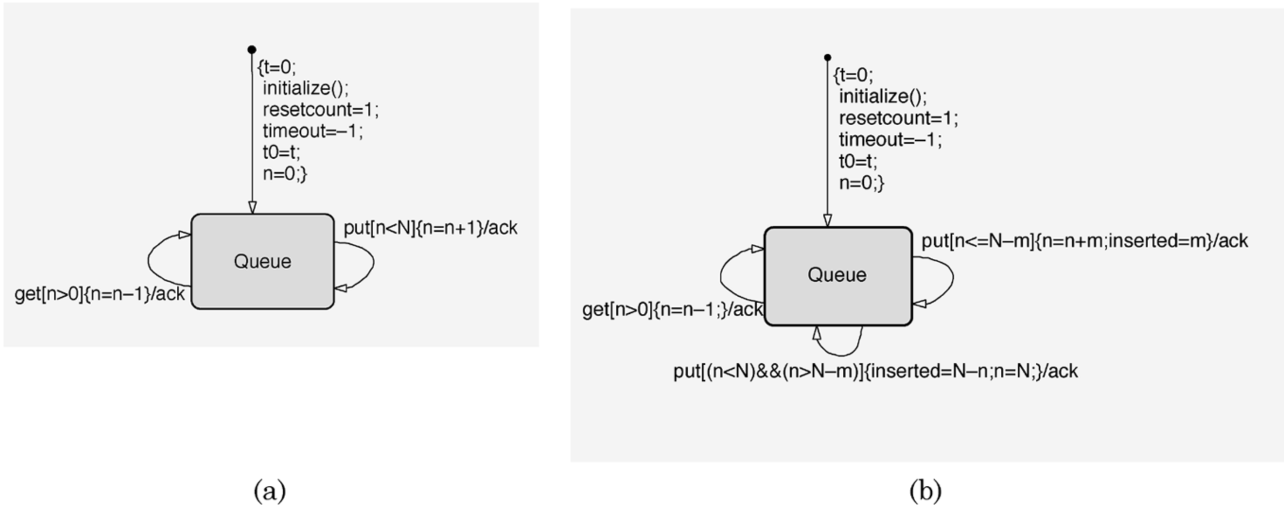


Fig. 14. Queue of size N represented by means of a unique parameterized GSMP state. (a) Input event *put* represents single incoming requests. (b) Input event *put* is parameterized as well to represent an arbitrary number (m) of simultaneous incoming requests.

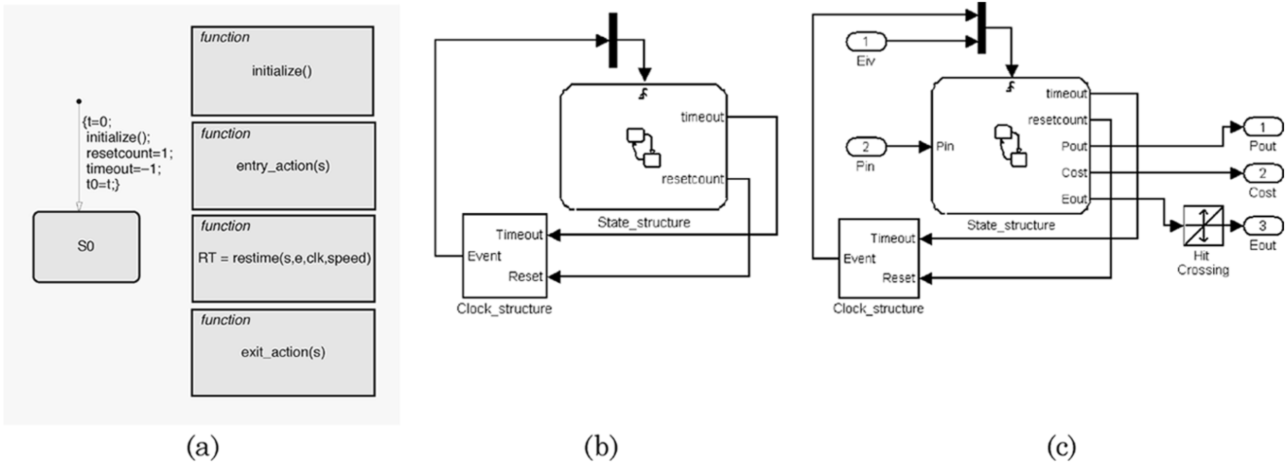


Fig. 15. Template of GSMP state. (a) State structure. (b) Clock structure and I/O interface. (c) Extended I/O interface.

5) *Definition of parameters*. Add data structures to represent metrics and parameters associated with state and event classes. Specify the scope of each parameter: local (if associated with states or unobservable internal events), input (if associated with input events), or output (if associated with observable events or metrics).
6) *Specification of the interface*. Add and connect input–output ports as illustrated in Fig. 15(c), where `Ein` is a generic input event, `Pin` is a generic input parameter, `Eout` is a generic output event, `Pout` is a generic output parameter, and `Cost` is a generic metric.
7) *Creation of state classes*. Create the `S` states by copying and renaming reset state `S0`. Assign unique names and identifiers to each state.
8) *Creation of state structure*. Add state transitions. For each transition specify the triggering event, the triggering conditions, the actions possibly associated with the edge traversal, and the output event possibly generated during edge traversal.
9) *Assignment of clock speeds*. Edit the `initialize` function and assign positive speeds to the clocks associated with active events. If event e is active in state s, a positive speed has to be assigned to `speeds[s][e]`.
10) *Assignment of residual-time distributions*. Edit the `entry_action` function and assign a residual-time distribution to each active event. This is done by selecting probability distributions from a Matlab library. The library can be easily extended to include arbitrary distributions.

Once GSMP models have been created for all system components, a top-level system description can be created by instantiating the components and creating connections among them. Also, standard Simulink components can be added at this level for monitoring system metrics and collecting execution traces.

No specific GUI has been developed for assisting the designer through the above-mentioned steps, since they can be easily performed using the built-in interface of MathWorks' Simulink and Stateflow toolkits. On the other hand,

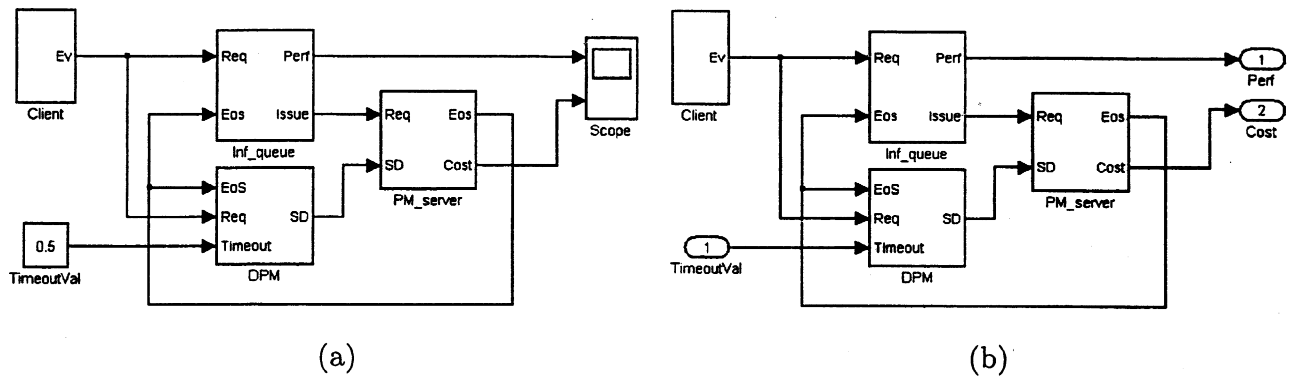


Fig. 16. Client–server system with timeout-based DPM. Independent parameters and dependent design metrics can be made controllable and observable either by means of `Constant` and `Scope` components (a), or by means of input and output ports (b).

a specific GUI could be used to enforce consistency with the DES/GSMP model.

E. Design Exploration and Optimization

Any DES/GSMP model specified in Simulink can be either directly simulated or inserted in a library for future instantiation. If the system is not autonomous, additional components have to be added to the system to generate input signals and events. A simulation run may provide (partial) traces of the system trajectory, timing behaviors of (some) design metrics, cumulative counts, and statistics. To this purpose, standard Simulink components can be connected to the output ports of the system to plot timing waveforms, compute event counts, and perform arbitrary runtime processing of design metrics.

The parameters of a DES/GSMP model represent system features (e.g., the power consumption of a given operating mode), design choices (e.g., the timeout to be used to shut down the system), and workload conditions (e.g., the average interarrival time of service requests generated by an exponential source). The Simulink model of a given system is an implementation of the (nondeterministic) functional relation between the configuration of independent parameters (representing the degrees of freedom of the design space and the workload conditions) and the configuration of design metrics. If we denote by X the configuration of independent parameters, by Y the configuration of design metrics, and by F the functional relation among them, each simulation run provides a *point estimate* of $Y = F(X)$ that is an estimate of the design metrics for fixed values of design parameters and workload conditions.

In case of nondeterminism, each point estimate may require multiple simulation runs performed under the same conditions, or a long simulation run with runtime evaluation of convergence criteria for the parameters of interest.

Point estimates can be repeatedly performed (for different configurations of design parameters) to perform design exploration and iterative optimization. Using the notation introduced above, design space exploration can be performed by sampling $Y = F(X)$. In many cases, exploration reduces to a sweep on a single dimension obtained by changing a single parameter while keeping all other parameters unchanged. Constrained optimization consists of changing the independent variables (within given ranges) in order to minimize/maximize the objective function (i.e., one of the design metrics) while satisfying constraints imposed to other metrics.

The inner loop of both design exploration and optimization involves: 1) tuning model parameters; 2) launching simulation; and 3) collecting and evaluating simulation results. Simulink provides three different ways for controlling simulation and collecting results: 1) using the GUI of Simulink; 2) using a Simulink module placed at a higher level of abstraction; and 3) using a programming language interface.

Using the GUI of Simulink is straightforward. Some of the parameters of the Simulink modules can be made directly accessible from the top-level schematic by means of `Constant` blocks and provided to the corresponding modules through specific input ports. Similarly, the timing behavior of the design metrics can be monitored by using one or more `Scopes`.

Example 11: The system of Fig. 16(a) is made of three components: a power-manageable server with deterministic service time and a single sleep state (`PM_server`), an infinite FIFO queue (`Inf_queue`), a nondeterministic source of service requests (`Client`), and a timeout-based PM (`DPM`). Assume that for given values of all model parameters (interarrival time of service requests, service time, wake-up time, etc.) we want to evaluate the effect of the timeout value to the power–performance tradeoff. This can be done from the top-level view of the system by setting the value of `TimeoutVal`, launching the simulation and monitoring the values of `Cost` and `Perf` plotted on the `Scope`.

Although the GUI is user-friendly and effective, it does not enable automation of parametric exploration and iterative optimization. To support automatic design-space exploration/optimization, input and output ports have to be added to the top-level schematic in order to make independent parameters and design metrics accessible through an external interface. This is shown in Fig. 16(b) for the system of Example 11. Once the external interface has been created, it

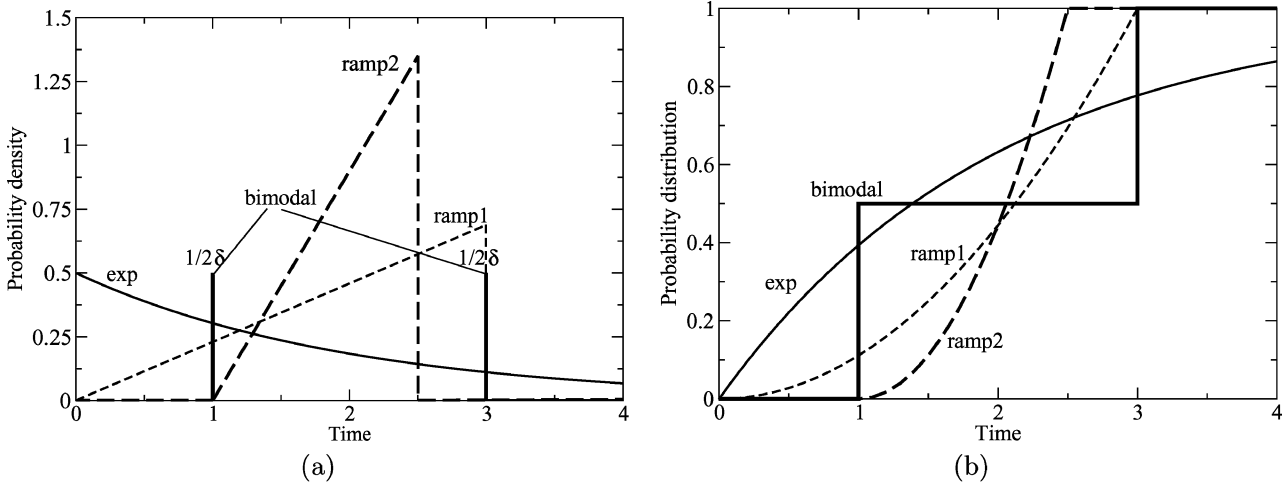


Fig. 17. Comparison of four different residual-time distributions with the same average value. (a) Probability density functions. (b) Probability distributions.

can be used to automatically invoke the evaluation of function $Y = F(X)$ from the inner loop of an exploration/optimization procedure, hereafter called *exploration procedure* for brevity.

The exploration procedure can be implemented either as an extra Simulink component (called `Exploration_Controller`), or as a Matlab function. The `Exploration_Controller` communicates with the system under exploration at run time, by means of input/output ports that complement those of the system: an output port for setting each independent parameter, an input port for capturing each metric of interest. The Matlab function, on the contrary, assigns values to the independent parameters, invokes a simulation run by means of the `sim()` command, and gets back from the simulator time-stamped traces of design metrics.

The key difference between exploration procedures implemented by Simulink modules and by Matlab functions is that Simulink modules may perform online tuning, while Matlab functions may only invoke batch simulation runs.

V. Applications

We present now the application of the theoretical analysis and of the simulator to practical real cases of power managed systems. We begin with a simple example, involving the computation of an optimum timeout, to exemplify the application of the exploration and optimization techniques. We continue with three other complex examples to stress the significance and novelty of the proposed technique.

A. System Shutdown

We consider in this section the client–server system of Fig. 16. The server has four internal states: `Waiting`, `Busy`, `Sleep`, and `WakingUp`. The power consumption is one in all states but `Sleep`, where it is negligible. Transitions to the `Sleep` state are triggered by an external `SD` event issued by the DPM when the system is `Waiting`. While shutdown transitions are instantaneous, wake-up transitions take three time units. The service time of the server, when active, is always 0.5.

The infinite FIFO queue between the client and the server has the main purpose of monitoring the waiting time of the incoming requests. The `Perf` metric provided by the `Inf_queue` component is nothing but the average waiting time.

Service requests are issued by a nonblocking client with a given distribution of interarrival times. In particular, we consider four different distributions with the same average interarrival time of two time units: exponential, bimodal, ramp1 (whose probability density grows linearly between zero and three time units and is null for larger times) and ramp2 (whose probability density grows linearly between 1 and 2.5 time units and is null outside that range). The four probability density and distribution functions are shown in Fig. 17.

We compare the power–performance tradeoff achieved by two DPMs:

- a *timeout-based DPM* that issues `SD` commands as soon as the system has been waiting for a given amount of time;
- a *randomized DPM* that issues nondeterministic `SD` commands whenever the system enters the `Waiting` state.

We study the effectiveness of both DPM strategies by performing parametric sweeps on the timeout value (`TimeoutVal`) and on the probability of issuing a `SD` command (`P(SD)`).

Simulation results for the exponential source are plotted in Fig. 18(a). If the system is always active (i.e., for `TimeoutVal` $>$ `10` or `P(SD)` $=$ `0`), the average service time is 0.5 and the average power consumption is one. For larger values of the shutdown probability (or for lower timeout values), the average power reduces at the cost of an increased waiting time. Interestingly, the Pareto curves provided by the two DPMs are almost coincident, meaning that the same tradeoffs can be achieved by both techniques.

Simulation results obtained for the bimodal input distribution, plotted in Fig. 18(b), are more interesting. Since the

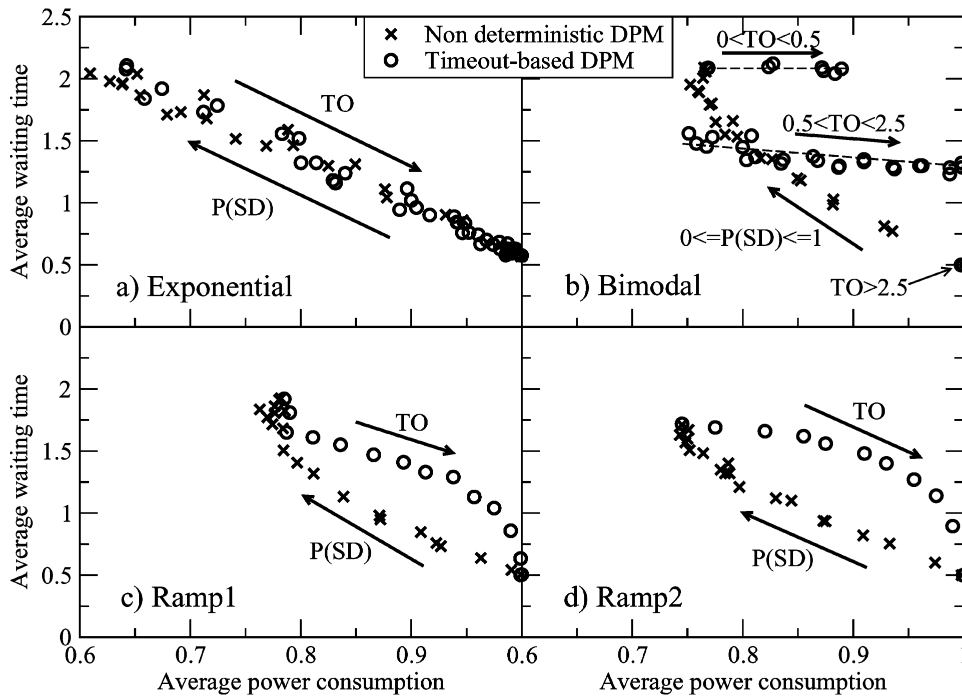


Fig. 18. Power performance tradeoff obtained for the system discussed in Section V-A by means of dynamic PMs based either on a timeout or on a randomized decision. The four plots refer to the residual-time distributions of service requests shown in Fig. 17. (a) Exponential. (b) Bimodal. (c) ramp1. (d) ramp2.

longest interarrival time between service requests is three, and the service time is 0.5, the longest idleness period is 2.5. If the timeout is greater than 2.5, the system never goes to `Sleep`. For timeout values lower than 2.5 and greater than 0.5, the system goes to sleep only when the interarrival time is three. Finally, for timeout values lower than 0.5, the system goes to sleep after each service. The power–performance tradeoffs achieved by changing the timeout value are represented in Fig. 18(b) by means of three disjoint curves: a horizontal segment for $0 < \texttt{TimeoutVal} < 0.5$, a diagonal segment for $0.5 < \texttt{TimeoutVal} < 1.5$, a point for $\texttt{TimeoutVal} > 2.5$.

All timeout values between 0 and 0.5 give rise to the same average waiting time, since the number of time-consuming wake-up transitions is always the same. On the other hand, the actual timeout value affects power consumption, since the longer the timeout, the larger the amount of energy wasted while waiting for the timeout to elapse.

Similar considerations hold for timeout values ranging from 0.5 to 2.5, but in this range the actual timeout has also a weak effect on performance. This is due to the cumulative service time of multiple enqueued requests that need to be serviced when the system wakes up. Whenever the cumulative service time is longer than 0.5 time units, the residual idle period corresponding to a interarrival time of three time units is shorter than 2.5. If the timeout value is longer than the actual idle period, the system does not go to `Sleep`, with a negative impact on power consumption and a positive impact on performance.

We remark that the left-most point of the segment obtained for $0.5 < \texttt{TimeoutVal} < 2.5$ represents a better solution (both in terms of power and in terms of waiting time) than any point in the horizontal segment obtained for $\texttt{TimeoutVal} < 0.5$. Hence, the horizontal segment is not on the Pareto curve. In particular, it is worth noting that the greedy policy corresponding to a null timeout does not provide an optimum tradeoff.

The tradeoff curve achieved by varying the shutdown probability $(\texttt{P(SD)})$ of a randomized DPM is also plotted in Fig. 18(b). Notice that it is a continuous curve and that it intersects the Pareto curve of the timeout-based shutdown. This means that the choice of the best DPM strategy depends on design constraints and on workload statistics.

Finally, the tradeoff points achieved with ramp-shaped probability density functions are reported in Fig. 18(c) and 18(d). We remark that they are substantially different from those obtained with exponential and bimodal distributions, and that timeout-based and nondeterministic DPM policies do not provide the same trade off. In particular, both for ramp1 and for ramp2, nondeterministic shutdown performs better than deterministic timeout.

The four clients with different interarrival time distributions were implemented starting from the template of a single-state GSMP, changing only the residual-time distribution. The sizable impact of the residual-time distributions on the power–performance tradeoffs shown in Fig. 18 motivates the need for the flexibility offered by GSMPs.

B. Multitasking Real-Time System

The simple server model of Section V-A hides all implementation details to retain only the service time. In most cases, the service is provided by a stack of software layers

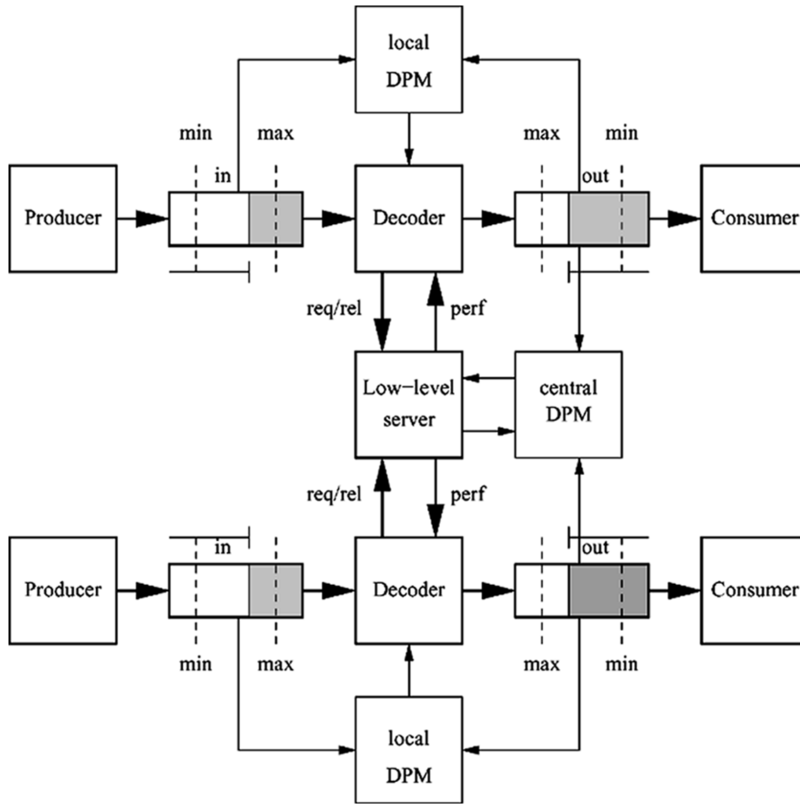


Fig. 19. Schematic representation of the multitasking computing system described in Section V-B, running two decoding processes.

running on top of a microprocessor platform. If the service time is modeled as a unique parameter, the model does not allow us to distinguish between single contributions to the global performance. Similarly, representing the workload of a system as a simple client that issues service requests with a given distribution is not always the best way for modeling a realistic workload.

In this section, we propose an advanced model of a multitasking computer system running concurrent real-time processes, namely, JPEG decoding tasks. The decoder is a software application executing on a computer system. The hardware platform and the operating system running on top of it are modeled as a unique low-level server that provides computational resources to the software processes, modeled as independent clients. Each process has its own input stream, generated by a producer, and output stream, used by a consumer. In practice, producer and consumer are clients of the software decoder, which in its turn has to request CPU time to the low-level (shared) server to accomplish its task. The service time depends on the interaction between the software application that implements the decoding task and the low-level platform that provides computational power. The application sets the amount of computation required to accomplish the task; the server sets the *computation rate* (i.e., the amount of computation per time unit) granted to the application. In particular, the computation rate viewed by each application varies over time depending on the number of concurrent tasks in the system. Assuming that the low-level server models a uniprocessor system working in multitasking, the computation rate provided to a single task is one, while the computation rate per task reduces to $0.9/N$ if there are $N > 1$ concurrent tasks in the system. Coefficient 0.9 accounts for a 10% overhead due to scheduling and context switch.

The distribution of the requests issued by producer and consumer imposes real-time constraints to each task. Input and output buffers are inserted between producer, decoder, and consumer to decouple real-time constraints from service time.

Fig. 19 shows the overall computing system with two decoding processes. Each process has its own producer and consumer and its own input and output buffers. Processes take input frames from their input buffers, request CPU time to the computation engine, perform a decoding task, release the CPU, and put the decoded frame into the output buffer. Local (i.e., distributed) and global (i.e., centralized) dynamic PMs (DPM) are also represented in Fig. 19. Local DPMs look at the state of the input/output buffers of a given process and possibly suspend its execution. The unique global DPM observes the state of the server and the number of frames in the output buffers of all active processes to decide when to shut down the server. For the sake of simplicity, only a few signals are represented in Fig. 19 while the details are outlined and discussed in the rest of this section.

The state structure of the low-level server is shown in Fig. 20(a). When active, the system may be `Waiting` or `Busy`. Transitions from `Waiting` to `Busy` are triggered by a CPU request input event (`req`) issued by a software process. Transitions from `Busy` to `Waiting` are triggered by any CPU release input event (`rel`) received when there is a unique

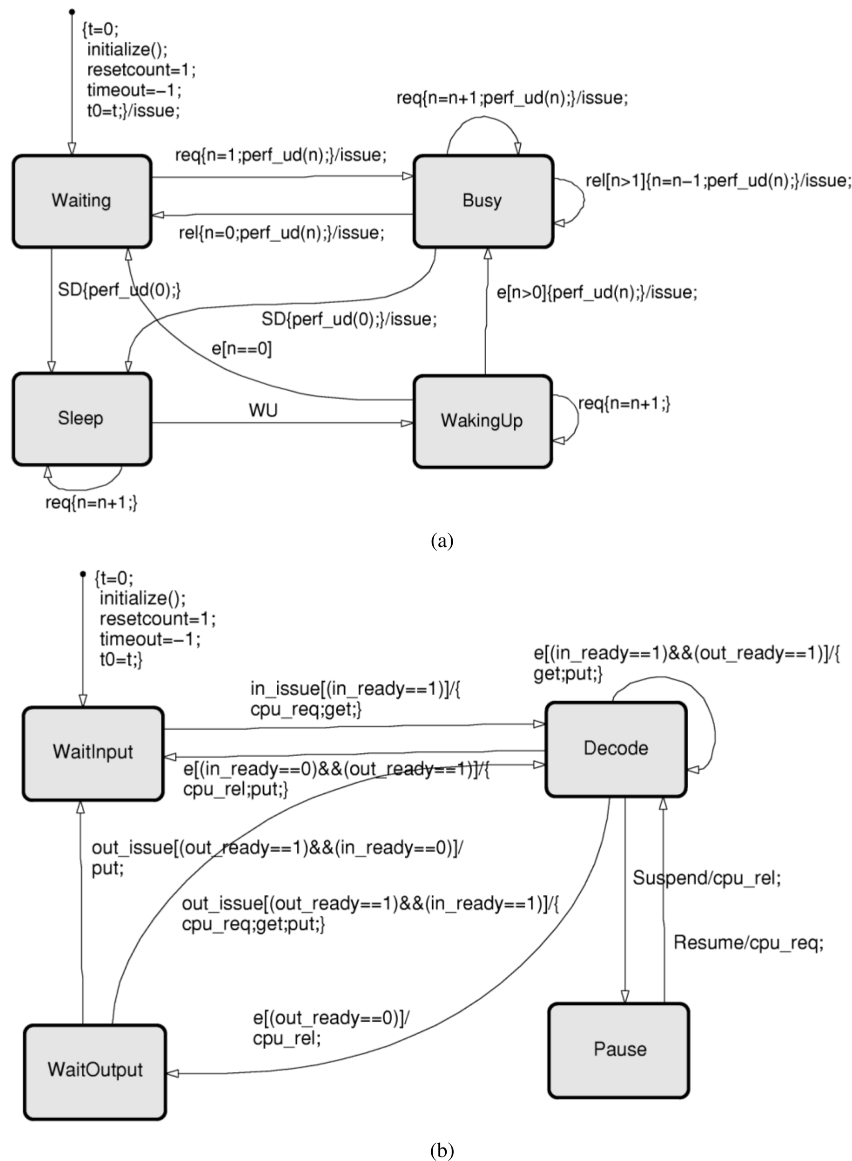


Fig. 20. State structures of: (a) a server representing a multiprogrammed computing system and (b) a software process acting as a client for the computing system it runs on.

task in the system. Since the server may serve multiple concurrent processes at the cost of decreasing the performance perceived by each process, the GSMP model of the server has a discrete set of busy states. We model the entire set as a unique `Busy` state parameterized by the number of tasks in the system (n), which is the runtime difference between the number of `req` and `rel` events received.

The server supports dynamic shutdown to a sleep state with negligible power consumption. Transitions to the sleep state take negligible time (i.e., they are modeled as instantaneous direct transitions) while wake-up transitions take finite amounts of time and energy, modeled by a `WakingUp` state. Shutdown and wake-up are triggered by input events SD and WU, issued by an external PM. There is a single internal event, active only in state `WakingUp`, that triggers the actual wake-up of the server. Notice that in our model the server can be shut down even if it is busy. In this case, the execution of all current tasks is suspended until next wake-up by setting performance to zero. Moreover, the system does not wake up until a WU event is received from the DPM. This means that several CPU requests may be received while in `Sleep` or `WakingUp` states. In order to keep track of the number of tasks to be serviced, `Sleep` and `WakingUp` states need to be parameterized using the same parameter n used for the `Busy` state.

The `Busy` state is the only state providing a nonnull performance. As for power consumption, we assigned power 0 to state `Sleep`, 1 to states `WakingUp` and `Busy`, and 0.5 to state `Waiting`.

The model of each software process [shown in Fig. 20(b)] has an inactive state (`Pause`) and three functional states: waiting for inputs (`WaitInput`), waiting for outputs (`WaitOutput`), and decoding (`Decode`). The process is allowed to start decoding a new frame if and only if the input buffer is not empty and the output buffer is not full. Otherwise, it has to wait for these conditions to be satisfied. The

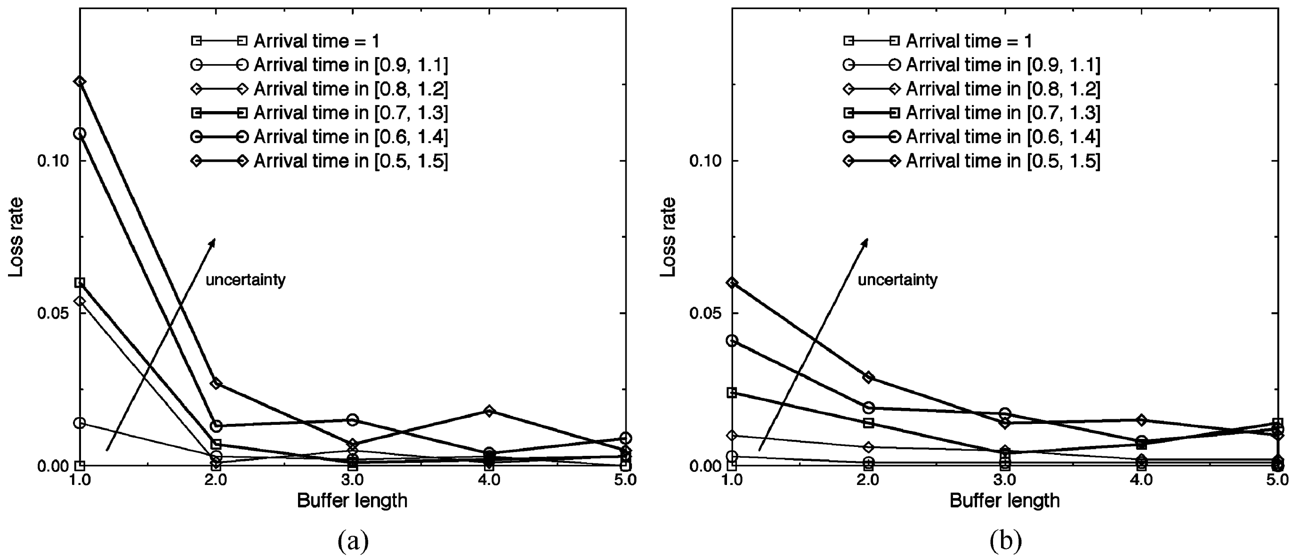


Fig. 21. Loss rate (i.e., probability of violating real-time constraints) for the example discussed in Section V-B as a function of buffer length. (a) Service time 0.5. (b) Service time 1.

CPU is requested/released whenever the process enters/exits the decoding state.

While a decoding task is initiated by external events (notifications of state changes of input/output buffers) its termination is triggered by an internal event representing the end of a service. However, the service time distribution depends on the runtime performance provided by the low-level server. Modeling this dependence requires the decoding state to be parameterized and a parametric event to be sent from the server to all clients to notify performance changes. It is also worth noting that the parameter is not discretized, since the decoding process does not know in advance the set of possible performance levels provided by the server. Hence, the decoder has a continuous-state model and the `Decode` state represents a continuous state set.

Finally, the decoder model has a `Pause` state possibly managed by an external PM by means of `Suspend` and `Resume` commands (events). When in `Pause`, the decoder releases the CPU.

We used our simulation environment to explore the effects of design choices, workload conditions, and DPM policies on the power–performance tradeoff.

1) Real-Time Constraints: We performed a first set of simulations to evaluate the minimum buffer size needed to meet input–output real-time constraints under uncertainty. We used a single decoding process with a periodic consumer, taking decoded frames from the output buffer every time unit, and a nondeterministic producer, with arrival times uniformly distributed between $1-x$ and $1+x$. In practice, x represents the degree of nondeterminism. The situation of a periodic consumer with a nondeterministic producer is typical of streaming media applications.

Fig. 21 plots the `loss rate` (i.e., the probability of loosing a frame because of real-time violations) as a function of buffer length for different degrees of nondeterminism. The same buffer length is used in input and output, so that frame loss may be caused either by a new frame produced when the input buffer is full, or by a decoded frame requested when the output buffer is empty. The plots in Fig. 21(a) and 21(b) refer to two different values of the performance of the decoder, providing service time 0.5 and 1, respectively. As expected, the loss probability decreases for larger buffers and increases with higher degrees of nondeterminism. Interestingly, reducing the performance of the decoder has a beneficial effect in terms of quality of service: for the same buffer length and input uncertainty, the loss probability is lower. This is because the longer service time contributes in decoupling output real-time constraints from input arrival times, increasing the effective buffer size. In fact, since the frame under process does not occupy any buffer, the longer its decoding time, the larger the average space left on the input and output buffers.

As a final remark, notice that loss rate is null regardless of the size of the buffer if the producer is deterministic.

2) Using Buffers for DPM: While the beneficial effects of buffers on real-time constraints are well known and intuitive, their impact on the power–performance tradeoff is worth a deeper discussion. In fact, large buffers offer degrees of freedom that can be fruitfully exploited for DPM purposes.

We first compute the baseline energy consumption of the system without DPM under the simple workload condition introduced in previous section: a single process decoding a frame in 0.5 time units, with a consumer requesting a decoded frame every time unit and a deterministic producer providing encoded frames at the same rate (i.e., with $x=0$). Since half of each time unit is spent to decode a frame and the other half to wait for the next one, the energy per time unit (i.e., per frame) is $0.5Cost(\mathtt{Busy}) + 0.5Cost(\mathtt{Waiting}) = 0.75$. The effectiveness of any DPM strategy has to be evaluated with respect to this baseline.

Before introducing buffer-based DPM policies, we show for comparison the effect of a timeout-based shutdown. In our model the decoding task is never paused, while the

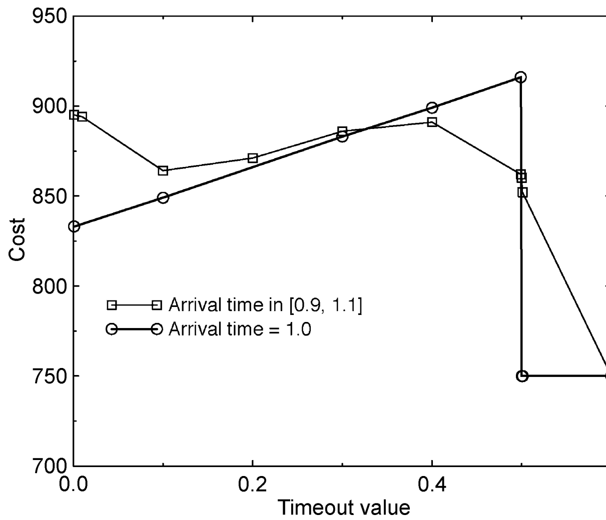


Fig. 22. Energy consumption (i.e., cost) as a function of the timeout used to shut down the CPU in the example in Section V-B. The dark line refers to a periodic workload; the light one refers to a nondeterministic workload with the same average arrival time.

DPM of the low-level server implements the timeout policy that issues an SD command whenever the server has been waiting for a given amount of time and a WU command whenever a new request is received. The bold line in Fig. 22 shows the energy consumption obtained for different timeout values by simulating the system for 1000 time units with a wake-up time of three time units. When the timeout exceeds the waiting time (0.5) the system never goes to sleep and its energy consumption is $1000 \times 0.75 = 750$. When the timeout is lower than 0.5, the server is periodically shut down. However, the idle time is shorter than the break-even time required to compensate wake-up cost, so that DPM is counterproductive. Moreover, the longer the timeout, the larger the wasted idle time and the higher the energy consumption. As for quality of service, buffers of length 2 are needed to avoid frame loss during wake-up.

Results obtained with nondeterministic arrival times uniformly distributed in [0.9, 1.1] are also shown in Fig. 22. In practice, nondeterminism has a smoothing effect on the energy versus timeout curve.

Since transitions to the sleep state are the only mechanism provided by the server to save power, Fig. 22 tells us that our workload conditions do not allow DPM, unless new degrees of freedom are introduced and exploited. Such additional degrees of freedom can be obtained by using input and output buffers exceeding the minimum size required to reach a given quality of service under real-time constraints (in particular, no buffers are required to meet real-time constraints in case of deterministic workload, as discussed in the last section).

Denoted by L (the length of the buffers) and by n_{frames} (the number of frames in a buffer), we define two thresholds (namely, $N_{\min}$ and $N_{\max}$ with $N_{\min} < N_{\max} < L$) to be used for DPM: the server is shut down whenever there are at least $N_{\max}$ decoded frames in the output buffer and $N_{\max}$ empty slots in the input buffer, while it is awakened whenever there are either less than $N_{\min}$ decoded frames in the output buffer of less than $N_{\min}$ empty slots in the input buffer.

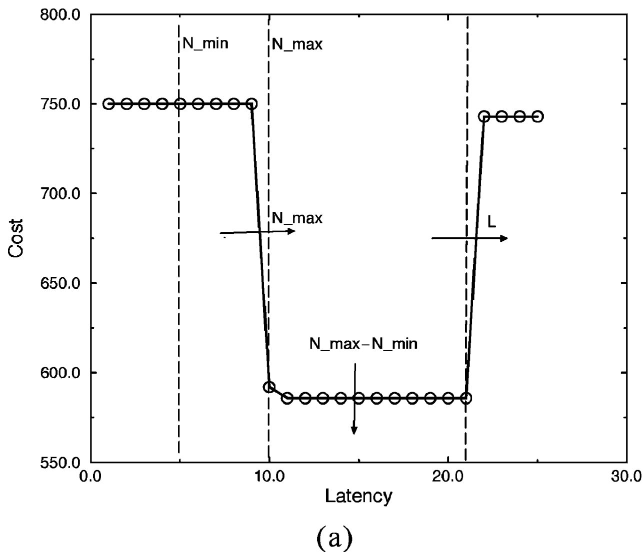
Notice that in our system, this policy can be implemented either by a global DPM that controls the server or by a local DPM that controls the process. The results provided in this section refer to a server-level (global) implementation. The DPM issues SD and WU commands for the server based on the observation of the I/O buffers of the process. When in sleep state, the server provides no performance, so that the process suspends execution without changing its state. Process-level (local) DPM policies will be discussed in the next section.

Consider, for instance, input and output buffers of size $L = 15$ with thresholds $N_{\min} = 5$ and $N_{\max} = 10$. If the consumer starts using the first frame as soon as it is decoded, the number of decoded frames in the output buffer never reaches the $N_{\max}$ threshold that would trigger a server shutdown. In fact, under our assumptions, each frame is decoded and consumed before the next one is produced. In the case of a deterministic producer and consumer working at the same rate, the effectiveness of buffer-based DPM depends on the *latency* of the consumer, i.e., on the relative delay between production and consumption. For our example, this is shown in Fig. 23(a). If the latency is lower than $N_{\max}$, the DPM never issues SD commands because of the insufficient number of frames in the output buffer. As the latency of the consumer exceeds $N_{\max}$ time units, more than $N_{\max}$ frames are produced, decoded, and stored in the output buffer before starting consuming them. Hence, at some point there are $N_{\max}$ frames in the output buffer and no frames in the input buffer, thus causing the DPM to issue an SD command. While the server is sleeping, the decoder does not work and encoded frames accumulate in the input buffer until the number of empty slots falls below $N_{\min}$. At this point, a WU command is issued or otherwise incoming frames will be lost. Input buffer saturation imposes an upper bound to the consumer latency required to enable DPM: $latency < L + N_{\max} - N_{\min}$.

When the latency is in the range compatible with DPM, the system enters a cyclic behavior whose periodicity depends on the distance between $N_{\max}$ and $N_{\min}$. The larger the difference $N_{\max} - N_{\min}$, the larger the number of frames in the output buffer ready to be used by the consumer and the slots in the input buffer available to store the frames provided by the producer while the server is sleeping. Since energy savings are proportional to the sleep time of the server, the larger $N_{\max} - N_{\min}$, the lower the energy.

The effect of the distance between SD and WU triggering conditions is shown in Fig. 23(b). Total energy is shown as a function of $N_{\min}$ for two different values of $N_{\max}$ (namely, 25 and 28). The effectiveness of DPM decreases as $N_{\min}$ gets closer to $N_{\max}$. In particular, for $N_{\min} = 24$ and $N_{\max} = 25$, the energy consumption exceeds the baseline value of 750. This is because the sleep time is lower than the break-even time of the server.

3) DPM With Multiple Processes: So far, we have considered a system running a single decoding process. In this section, we deal with multitasking. Although we could specify and simulate a large number of concurrent processes with nondeterministic workloads, for the sake of simplicity



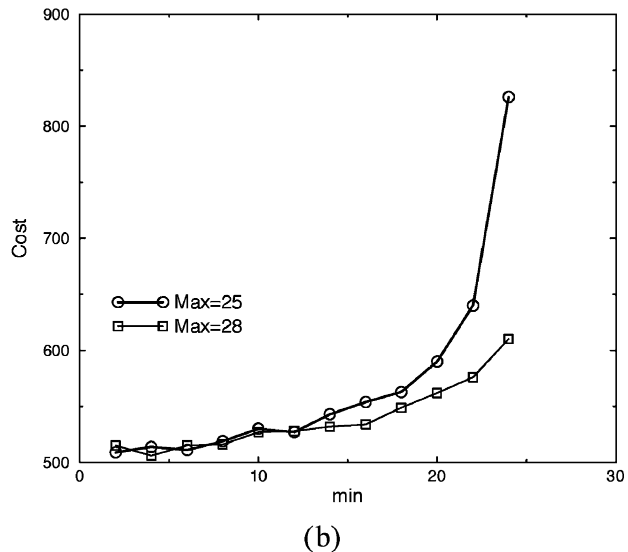


Fig. 23. Energy consumption (i.e., cost) of the system discussed in Section V-B with centralized buffer-based DMP and a single periodic workload. (a) Dependency on the latency of the consumer. (b) Effect of the upper and lower buffer thresholds used by DPM.

we discuss only a simple example consisting of two identical processes with deterministic workloads working at the same frame rate. Moreover, we assume that encoded frames can be directly taken from memory, so that the producer does not need to be modeled.

We implemented and simulated two DPM strategies.

1) *Centralized buffer-based DPM*: the DPM of the server looks at the output buffers of the two tasks. A SD command is issued if and only if there are more than $N_{\max}$ decoded frames in each output buffer. A WU command is issued whenever the number of decoded frames in either of the output buffers falls below $N_{\min}$. Local DPMs are not used.
2) *Distributed buffer-based DPM with centralized timeout*: the centralized DPM implements a timeout that issues SD commands for the server when it has been waiting for a given amount of time, and WU commands when a new request is received from a process. Local DPMs implement a buffer-based policy looking only at the output buffer: each process is paused when the number of decoded frames in its output buffer exceeds $N_{\max}$, while execution is resumed when there are no more than $N_{\min}$ decoded frames left. Since each process releases the CPU when in `Pause`, the low-level DPM may take advantage of process-level DPM.

Typical execution traces are shown in Fig. 24 for the two DPM policies (simply called *distributed* and *centralized*) working under the same operating conditions: frame rate 1, service time 0.5, $N_{\max} = 15$, $N_{\min} = 3$. The timeout of the server is zero, while the latencies of the consumers of the two processes are 80 and 95 time units. In the following, we will use the term *latency skew* to denote the difference between the starting time of the consumers of different processes. Although the two DPM policies are very similar, they give rise to fairly different traces.

Let us first look at the trace obtained with distributed DPM. At the beginning, both processes start decoding while their consumers are inactive. Since the two processes work concurrently, the performance perceived by each of them is $0.9/2 = 0.45$. Decoded frames accumulate in the output buffers until $N_{\max}$ is reached. At this point, both processes are suspended by their local DPM, the server is released, and the low-level DPM issues an SD command that puts the server to `Sleep`. Then nothing happens until the first consumer starts taking decoded frames from the output buffer, causing the number of frames in it (bold curve in the bottom graph) to reduce to $N_{\min}$. At this point, the local DPM resumes computation, the process sends a CPU request, the low-level DPM issues a WU command and the server goes to the `WakingUp` state, where it stays for three time units. While waking up, the server provides no performance, so that all frames in the output buffer are consumed before starting decoding new frames. When the server goes active, its performance is one, since there is a single task in the system. The peak performance of the server is exploited to decode frames at a rate much higher than that of the consumer, causing the output buffer to reach $N_{\max}$ in only 18 time units.

The second process has a similar behavior, but the latency skew of 15 cycles completely decouples their activities, so that they never compete for computational resources. This can be seen on the top graph, which shows that the effective performance provided by the server while active is always one, meaning that there is a single process at the time in the system.

The effect of centralized DPM is completely different, since it tends to realign the two processes. In this case, in fact, the two processes are never paused, so that they never release computational resources. When the server goes to sleep the software processes suspend their execution just because the effective performance provided by the server

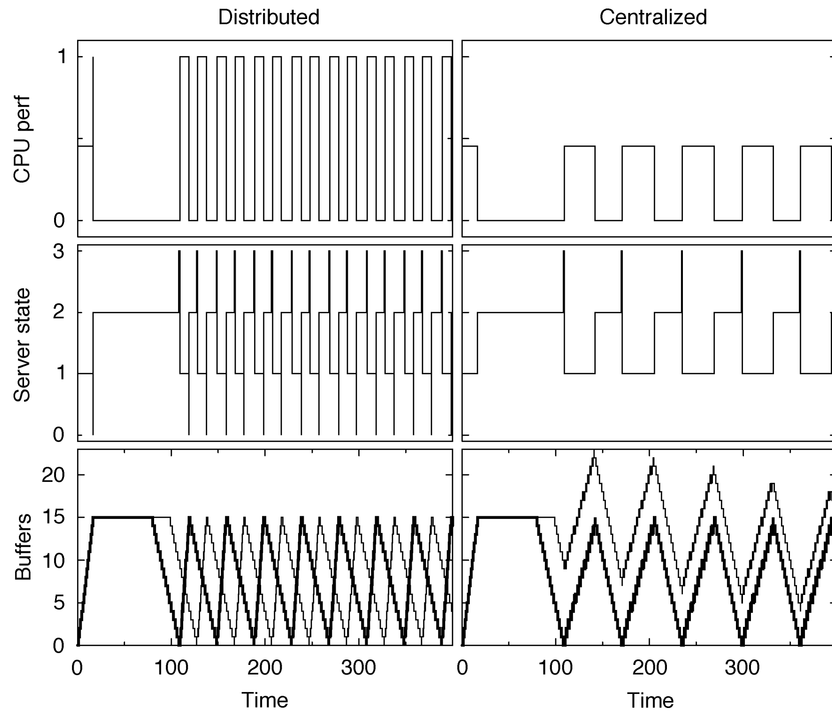


Fig. 24. Execution traces obtained by simulating the multitasking system in Section V-B3 with distributed and centralized buffer-based DPM strategies. For each case, three plots are reported: instantaneous performance provided by the server to each active task, state of the server ($0 =$ `Waiting`, $1 =$ `Busy`, $2 =$ `Sleep`, $3 =$ `WakingUp`), and state of the output buffers of the two decoding processes.

is null. However, when the server is awakened because of the $N_{\min}$ threshold reached by the first process, both processes are still in the system and resume computation (with performance 0.45) until both output buffers reach the $N_{\max}$ threshold. There are two main differences with respect to the previous case: 1) the two processes work together regardless of the initial skew, so that the effective computation rate is always 0.45 and 2) the number of frames in the output buffer of the second process may be much higher than $N_{\max}$ (even if the overshoots tend to disappear because of the wake-up time of the server, that makes the falling edges longer than the rising ones).

The energy savings provided by the two DPM strategies are discussed referring to Fig. 25, which reports the energy consumption per frame (and the frame loss probability) as a function of the latency skew between the consumers of the two processes. The skew is zero when the two processes are synchronized. In order to show on the same graph results obtained with different frame rates, latency skew values are normalized to the periodicity of the corresponding execution traces. Results are reported for centralized DPM, and for distributed DPM associated with low-level timeout of zero and ten time units.

First of all, we remark that the energy efficiency of centralized DPM does not depend significantly on the latency skew, nor on the frame rate. The independence from latency skew is due to the nature of centralized DPM, which tends to realign the two processes (as observed in Fig. 24), while the low dependence on frame rate demonstrates the effectiveness of centralized DPM. Without DPM, the energy per frame would be 0.725, 0.9125, and 2.9125 at 0.8, 0.5, and 0.1 frames per time unit, respectively. Notice that all simulations were performed assuming a service time of 0.5 time units and a power consumption of one when busy. Hence, the energy per frame cannot be lower than 0.5. The reason why the energy per frame obtained with centralized DPM is always above 0.55 is twofold: 1) waking up the server has a nonnegligible cost and 2) with centralized DPM, the two processes work simultaneously at a cost of a 10% performance penalty (due to context switch) that results in a 10% energy overhead. We also remark that the energy per frame for at a rate of 0.8 frames per time unit is 0.725, that is lower than the baseline of 0.75 computed in Section V-B2. This is due to multitasking, which reduces the energy overhead paid by the server while waiting for a new task.

As for quality of service, the loss probability is always null for frame rates of 0.1, while it is up to 0.016 for higher frame rates. In general, the loss probability depends on the difference between the time required to wake up the server and the time required to consume the $N_{\min}$ frames available on the output buffer. The shorter the wake-up time and the higher $N_{\min}$, the lower the loss probability. For centralized DPM, the rising and falling ramps observed for frame rates 0.5 and 0.8, are due to transient effects that occur only before the latency skew has been completely compensated by the DPM.

When the consumers of the two processes are aligned (latency skew 0), the centralized DPM is equivalent to the distributed DPM with timeout 0. In fact, the curves on the second graph of Fig. 25 start from the same values observed on the first graph. The energy efficiency changes, however,

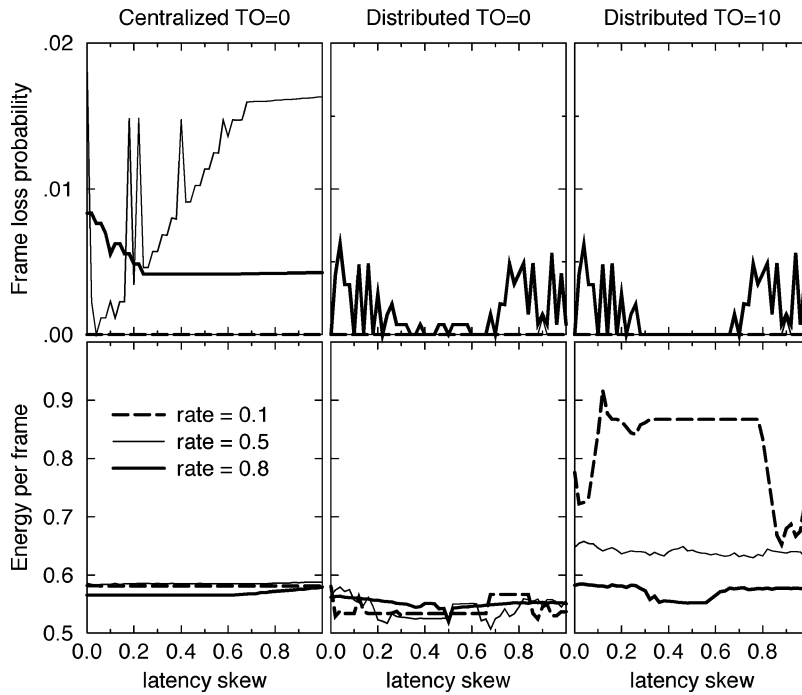


Fig. 25. Energy efficiency and loss probability plotted as functions of the latency skew between the consumers of two concurrent decoding processes. Plots have been obtained by simulating the effects of the DPM strategies described in Section V-B3: a centralized DPM, a distributed DPM without timeout, and a distributed DPM with a centralized timeout.

for latency skews greater than zero. This is because of the twofold effect of misalignment that can be seen in Fig. 24: when the two processes have disjoint computational needs, the performance provided to each task increases from 0.45 to 1, and the number of shutdown/wake-up transitions doubles. While the performance increase improves energy efficiency by reducing the effective CPU time per frame, the larger number of state transitions impairs energy efficiency because of the wake-up energy. In our case study, the beneficial effect of increased performance dominates, so that the energy per frame provided by the distributed DPM with zero timeout is lower than that provided by centralized DPM.

Finally, the third graph shows the effect of a low-level timeout greater than zero used together with distributed DPM. In all our cases, the timeout impairs energy efficiency by wasting idle time. However, the effect strongly depends on the rate of the consumer: the lower the frame rate, the higher the cost of each frame. This counterintuitive result can be explained by thinking that the best energy efficiency (i.e., 0.5) would be obtained by a system keeping the CPU always busy. If the consumer works at high frame rates, a large number of frames are consumed during decoding, making it hard for the output buffer to reach the $N_{\max}$ level that triggers a server shut down. Hence, the number of frames decoded before going to sleep is higher at higher frame rates. Since each shutdown/wake-up cycle has a cost in terms of both wasted idle time (i.e., timeout) and wake-up energy, if this cost is distributed over a larger number of frames, the energy overhead per frame is lower.

For both distributed DPM policies, the loss probability is nonnull only at the highest frame rate and it has a noisy dependence on the latency skew.

From the above discussion, the best results seem to be provided by distributed DPM without low-level timeout. We remark, however, that the validity of this conclusion depends on system parameters. Systems with larger I/O buffers would have no quality loss, while systems with larger scheduling overheads or lower wake-up costs would take greater advantage of the decoupling provided by distributed DPM.

C. Sensor Networks

Sensor networks have been introduced in Section II-C2 as examples of systems composed of a large number of interacting power-manageable components. In general, each sensing node is a reactive component that reacts to a sensed event by performing some local processing and/or by notifying the event to a base station. Hence, it has at least two operating modes: passive (i.e., sensing) and active (i.e., processing/storing/transmitting).

Although complex power-manageable sensors may be conceived, in this section we consider a network composed of simple sensing elements that may only decide whether or not to react to an event based on the current state of their batteries. Moreover, we consider that the power consumption in passive mode is negligible, while event processing and notification have a fixed cost in terms of energy. We present and discuss simulation results showing how the energy efficiency of the network depends on local runtime decisions and on the global organization.

We consider a network of sensors distributed on a grid over a region of size $R \times C$. For the sake of simplicity, we assume that each sensor covers a square zone of size $L \times L$ and we denote by d the linear distance between contiguous sensing nodes and by N the number of sensing nodes (i.e., the *size*

of the network). If $d = L$ the sensing zones of the sensors do not overlap and the network has the minimum number of nodes needed to guarantee complete coverage of the target region: $N = RC/L^2$. If $d < L$, the sensing zones do overlap and events occurring in overlapping regions may be sensed by more than one sensing node. If $d > L$, on the contrary, sensing zones are not contiguous and the target region is only partially covered by the sensor network. The ratio between the sum of the areas of the sensing zones of all sensing elements and the area of the target region provides a measure of the average *coverage* provided by the network, defined as the average number of sensors that cover each target point

$$S_{\text{target}} = RC$$
$$S_{\text{covered}} = \frac{R}{d}\frac{C}{d}L^2$$
$$\text{AreaCoverage} = \frac{S_{\text{covered}}}{S_{\text{target}}} = \frac{L^2}{d^2}.$$

If we take into account the energy budget E of each sensing element and the energy spent for each detection ΔE, under the assumption of negligible power consumption in sensing mode, we obtain the detection capacity of each sensor, that is, the number of events it can sense before running out of power

$$n = \frac{E}{\Delta E}.$$

The product between area coverage and detection capacity provides a measure of the maximum number of events occurring on the same point that can be properly detected

$$\text{LocalEventCoverage} = \frac{E}{\Delta E}\frac{L^2}{d^2}. \quad (5)$$

Notice, however, that the detection of an event occurring at a given point impairs the detection capacity on all points covered by the same sensors. In particular, the average number of detectable events per area unit is given by:

$$\text{AverageEventCoverage} = \frac{E}{\Delta E}\frac{1}{d^2} \quad (6)$$

According to (6), the average event coverage may be improved in three equivalent ways: 1) by reducing the space between sensors d; 2) by increasing the energy budget of each element E; and 3) by improving energy efficiency (i.e., by reducing ΔE). The size of the field of view of each sensor (L) does not affect the average event coverage, but it does affect the maximum local event coverage.

The actual exploitation of the detection potential expressed by (5) and (6) depends both on the network organization and on the event distribution. In the following, we analyze the energy efficiency of four different implementations with the same global energy budget.

1) *Static priority* ($d = 1$, $L = 2$, $E = 10$): a sensor network with $L = 2d = 2$, $E = 10\Delta E$, and fixed lexicographic priorities used to decide which sensor has to take care of events occurring in points covered more than once [Fig. 26(b)].
2) *Dynamic priority* ($d = 1$, $L = 2$, $E = 10$): a sensor network with $L = 2d = 2$, $E = 10\Delta E$, and dynamic priorities based on residual energy: among the sensing elements that may sense a given event, the one that actually takes care of detection is the one with the higher level of residual energy. Fixed priorities are used only to decide among sensors with the same energy left [Fig. 26(b)].
3) *Nonoverlapping* ($d = 1$, $L = 1$, $E = 10$): a sensor network with $L = d = 1$ and $E = 10\Delta E$ [Fig. 26(c)].
4) *Nonoverlapping* ($d = 2$, $L = 2$, $E = 40$): a sensor network with $L = d = 2$ and $E = 40\Delta E$ [Fig. 26(e)].

The local event coverage provided by each network is represented in Fig. 26. White regions have a local event coverage of 10, light shaded regions of 20, dark shaded regions of 40.

Priorities among overlapping sensors are handled by means of a simple mechanism based on timeouts and local notifications: each sensor reacts to a sensed event by setting a timeout that is inversely proportional to its priority. If a service notification arrives while waiting for the timeout, the sensor resets its timeout and goes back to the sensing mode waiting for other events. If the timeout elapses without incoming notifications from other sensors, it sends a service notification to its neighbors and starts processing the event.

Static priorities are implemented by assigning static timeouts to each sensor, growing from the upper left to the bottom right corner of the grid. Dynamic, energy-aware priorities are implemented by means of an additional delay proportional to the number of events already processed by the sensor. Instances of the sensor model are specialized by means of input parameters specifying its position (x, y), the size of its field of view (L), the energy budget (E), and the static delay $(T0)$.

The workload of the network is modeled by a GSMP component that generates intrusion events with a given spatial and temporal distribution. Generation of an intrusion event consists of: 1) generating a x, y pair to be assigned to two output signals and 2) issuing an output triggering event (`intrusion`). Spatial coordinates and triggering events are provided to all sensors. When the triggering event is received, each sensor samples the x, y input signals representing the point of intrusion and it evaluates whether it falls within its field of view. If this is not the case, it does nothing; otherwise, it enters the timeout state possibly leading to event processing. In practice, intrusions are modeled as parametric events distributed to all sensing elements, while locality checks are demanded to each element.

We modeled the four sensor networks described above and we performed a first set of simulations using random intrusion events with constant rate and uniform spatial distribution. Then we compared the event coverage provided by the networks in terms of three quality metrics computed on a reference region.

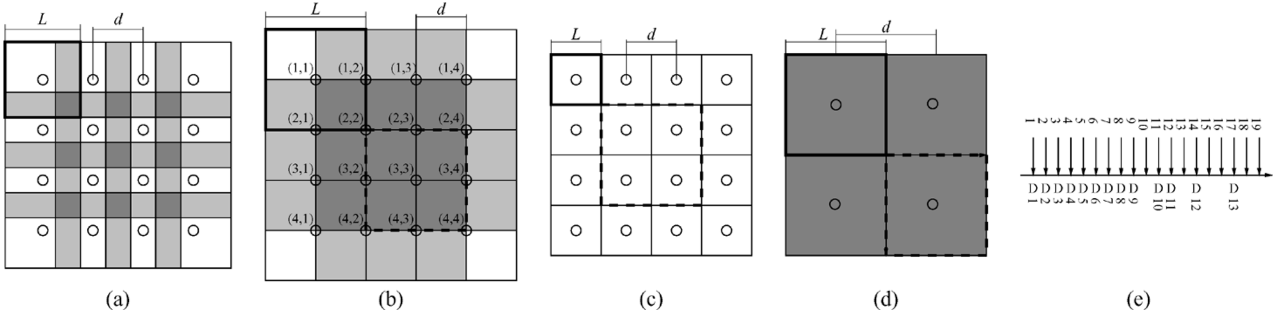


Fig. 26. Event coverage provided by different sensor networks with the same energy budget. (a) Event coverage of a generic network with partially overlapping fields of view. (b) Event coverage of sensor networks 1 and 2 of Section V-C. (c) and (d) Event coverage of the sensor networks 3 and 4 of Section V-C with nonoverlapping fields of view. (e) Symbolic representation of a sequence of intrusions and detections.

Table 1
Sample Average and Standard Deviation of the Quality Metrics Defined in Section V-C, Evaluated for Four Different Sensor Networks With the Same Energy Budget

Network No.	N_{FMI} Avg	N_{FMI} StDev	N_{LDI} Avg	N_{LDI} StDec	N_{DI} Avg	N_{DI} Stdev
1	31.01	4.21	54.38	8.42	41.22	4.50
2	38.62	4.00	39.86	5.57	39.12	4.87
3	33	4.38	49.3	4.84	40	0
4	41	0	40	0	40	0

- *First missed intrusion* (N_{FMI}), that is, the first intrusion that cannot be detected, since none of the sensors covering the intrusion point have sufficient residual energy.
- *Last detected intrusion* (N_{LDI}), that is, the last intrusion occurring in the reference region before all sensors covering that region finish their energy budget.
- *Total number of detected intrusions* (N_{DI}), that is, the total number of intrusions detected in the reference zone before all sensors covering that zone finish their energy budget.

Fig. 26(a) shows a symbolic representation of a sequence of enumerated intrusions and detections. Intrusions are represented by vertical arrows, while detections are represented by capital D. The quality metrics computed on the trace of Fig. 26(a) are $N_{\mathrm{FMI}} = 10$, $N_{\mathrm{LDI}} = 17$, and $N_{\mathrm{DI}} = 13$. On small regions covered by a single sensor, N_{LDI} and N_{DI} always take the same value, while N_{FMI} is always equal to $N_{\mathrm{DI}} + 1$. On larger regions covered by multiple (possibly overlapping) sensors, all quality metrics are significant. For our experiments we used a reference region of size 4, shown in Fig. 26(b), (c), and (d) by means of a dashed box, representative of the coverage guaranteed by the inner elements of the network.

Each simulation was repeated 100 times in order to compute sample average and standard deviation of the three quality metrics. Results are reported in Table 1. Networks 2 (i.e., dynamic priority with $L = 2$, $d = 1$, $E = 10\Delta E$) and 4 (i.e., nonoverlapping with $L = d = 2$ and energy budget $E = 40\Delta E$) are more reliable, since they detect the higher number of intrusions before the first miss.

The performance of the four sensor networks was also tested against deterministic intrusion patterns, representative of possible malicious attacks occurring on the reference 2 $\times$ 2 region. The intrusion patterns are schematically represented in Fig. 27(a).

- *Pattern A*: a sequence of intrusions occurring in the same point.
- *Pattern B*: a sequence of interleaved intrusions iteratively occurring on the four quadrants.
- *Pattern C*: four series of intrusions occurring on each quadrant in sequence.
- *Pattern D*: a series of intrusions alternatively occurring on the first and fourth quadrants.
- *Pattern E*: a series of intrusions alternatively occurring on the second and fourth quadrants.
- *Pattern F*: a series of intrusions distributed along a diagonal line entering the square region from the upper left corner.
- *Pattern G*: a series of intrusions distributed along a diagonal line entering the square region from lower right corner.
- *Pattern H*: a series of intrusions distributed along a vertical (or horizontal) line across the square region.

Intrusion patterns F, G, and H are larger than the reference region, since the effective event coverage on the region of interest may be affected by the detection of intrusions occurring outside of it. In particular, this is the case whenever the fields of view of some sensors cross the boundaries of the reference region, as in networks 1 and 2.

All combinations of sensor networks and intrusion patterns were simulated to evaluate two metrics: the number of intrusions in the reference region detected without errors (i.e., $N_{\mathrm{fmi}} - 1$) and the area of the reference region left uncovered after the first $N_{\mathrm{fmi}} - 1$ intrusions. Comparative results are reported in the bar graphs of Fig. 27(b) and 27(c). The superior quality of network 2, with overlapping fields of view and energy-driven priorities, is apparent: it always provides the largest event coverage, while leaving a small area uncovered at the end of the experiment. Notice that for patterns B, D, and E, network 1 leaves an uncovered area smaller than network 2. However, this is due to the lower number of intrusions detected by network 1. In fact, each simulation

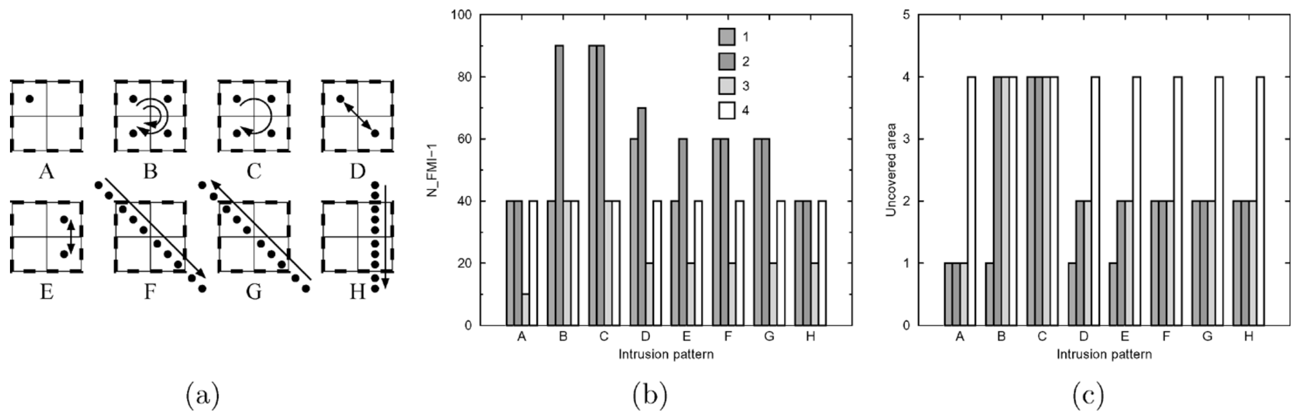


Fig. 27. (a) Deterministic patterns of intrusions and their effects on the four sensor networks of Section V-C, represented in terms of: (b) number of intrusions detected without errors $N_{\rm fmi} - 1$ and (c) area of the region left uncovered after the first $N_{\rm fmi} - 1$ detections.

was stopped after the first missed intrusion and the uncovered area evaluated at that point.

D. The Intel Xscale Processor

Both the supply voltage and the speed of the Intel Xscale processor core can be dynamically adjusted within given implementation-specific ranges. Hence, when active, the processor may operate at different power–performance levels, associated with a continuous set of states in the GSMP model. The supply voltage provided to the core determines the upper bound for the corresponding clock frequency. The best power–performance tradeoff is achieved when the processor operates at the lowest supply voltage compatible with the target performance. However, voltage scaling and clock adjustment take a finite amount of time and cannot be performed simultaneously. This makes DPM decisions nontrivial and motivates the existence of suboptimal operating states, characterized by a supply voltage higher than the minimum value required to sustain the operating frequency.

We refer to a prototype implementation of the Xscale processor [25] with supply voltage ranging from 0.70 to 1.65 V, and clock frequency ranging from 50 to 800 MHz. The maximum power consumption is 900 mW. Frequency adjustment takes a fixed amount of time (around 30 μs) to relock the clock and requires the processor to suspend execution. Voltage scaling, on the contrary, can be performed during execution and takes a variable amount of time, determined by the maximum slew rate of the supply voltage (around 4 mV/μs).

We denote by $F_{\max}(V)$ the maximum frequency sustained by supply voltage V, and by $V_{\min}(F)$ the minimum supply voltage required to sustain clock frequency F. During clock adjustment, the supply voltage has to be high enough to sustain the higher clock frequency. Suppose, for instance, that the microprocessor core has to switch from F_0, $V_0 = V_{\min}(F_0)$ to F_1, $V_1 = V_{\min}(F_1)$. If the target frequency F_1 is lower than the original one F_0, then frequency adjustment has to be performed before voltage scaling. On the contrary, if the target frequency F_1 is greater than the original frequency F_0, then voltage scaling has to be performed first to sustain frequency adjustment. In both cases, no performance is provided during frequency adjustment, while during voltage scaling the processor core operates at $F = min\{F_0, F_1\}$.

1) GSMP Model: We model the active states of the Xscale processor by means of four parameterized states (`Waiting`, `Busy`, `TuningVdd`, and `TuningF`), as depicted in Fig. 28. Transitions between `Waiting` and `busy` states are triggered by incoming CPU requests and releases (corresponding to external events `req` and `rel`). Transitions to the two tuning states are triggered by a unique parameterized external event `tune` representing DPM commands. Parameters associated with the DPM command are the target supply voltage (`inVdd`) and clock frequency (`inF`), that are sampled by the Xscale model when the `tune` event is received. Since the effect of the DPM command depends on the current values of `inVdd` and `inF`, they are used to condition state transitions and tuning times (computed by the `tuningTime()` function before entering the tuning state `TuningVdd`). Finally, transitions from the tuning states to `Busy` and `Waiting` are triggered by the internal event `e`, whose residual time depends on the actual tuning time.

All the states depicted in Fig. 28 are parameterized in terms of supply voltage `Vdd` (defined in the range [0.7, 1.65]) and clock frequency `F` (defined in the range $[50\ \text{MHz}, F_{\max}(\texttt{Vdd})]$). In addition, parameter n is used to represent the number of tasks simultaneously processed by the CPU. For systems with a single blocking client, parameter n is never greater than one.

The performance provided by the CPU is determined based on the current state, on the clock frequency, and on the number of tasks in the CPU. In particular, the effective clock frequency perceived by each task is recomputed upon state changes by the `updatePerf()` function as $\text{Perf} = F$ for $n = 1$, and $\text{Perf} = 0.9F/n$ for $n > 1$. The power consumption is scaled by Vdd^2 and F, assuming a maximum power consumption of 0.9 W.

In this example, we decided not to use the inactive states of the Xscale processor, in order to focus only on voltage scaling and frequency adjustment. Accordingly, inactive states are not shown in Fig. 28 for the sake simplicity.

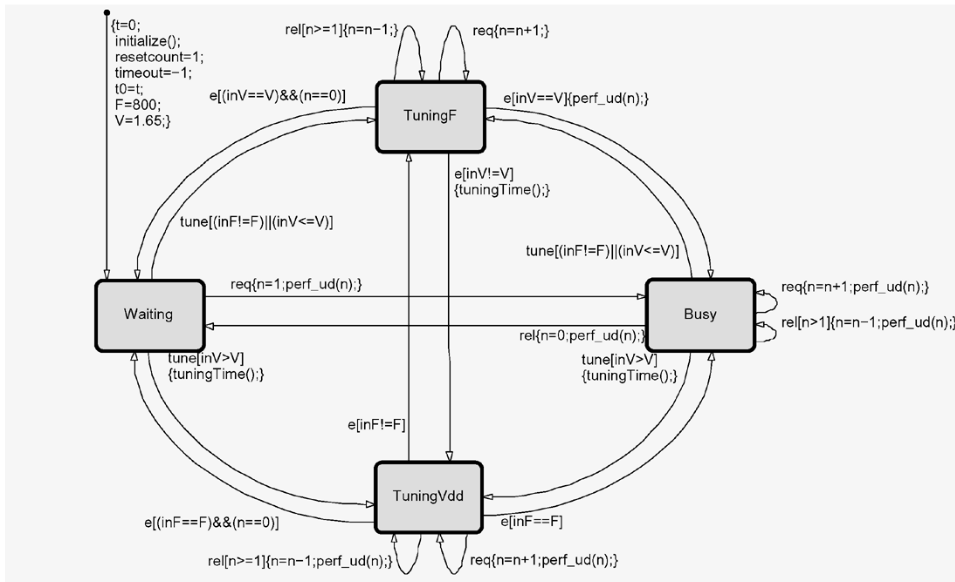


Fig. 28. State structure of the GSMP model of the active mode of the Xscale processor.

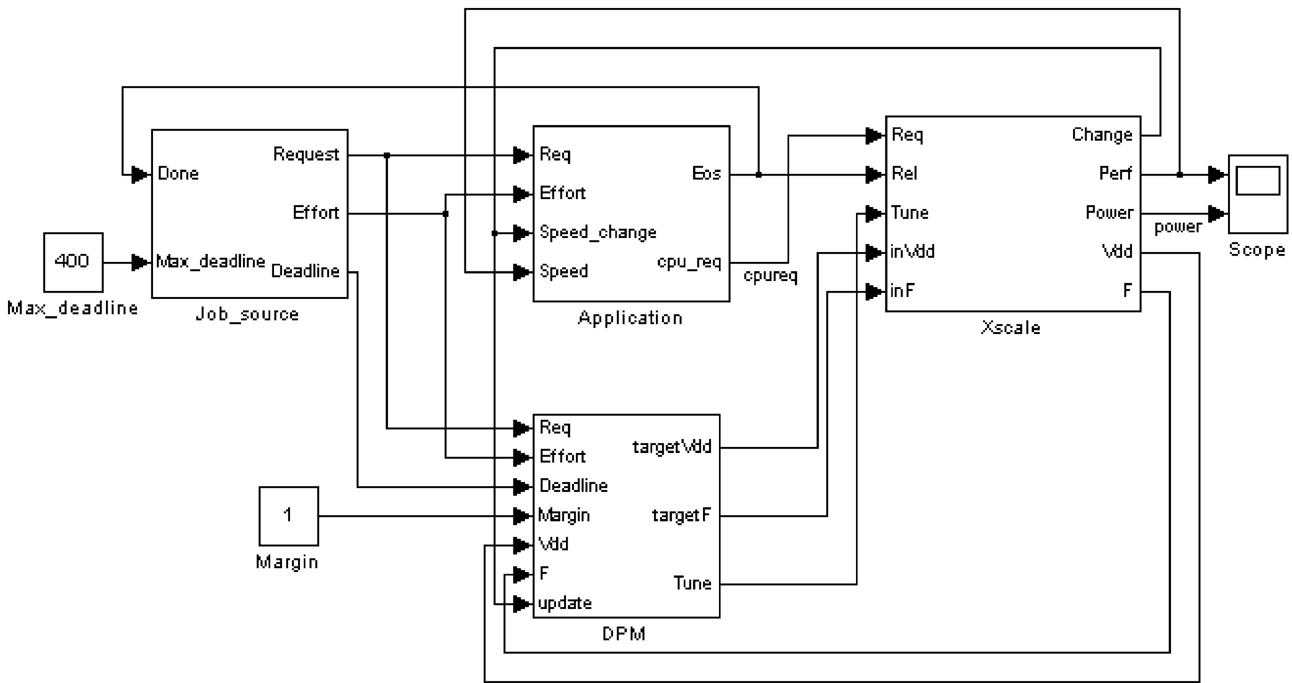


Fig. 29. Top-level schematic of a HW/SW system based on the Xscale processor.

2) Workload: The Xscale processor is used as the core of a simple HW/SW system that executes tasks requested by a single blocking client. Referring to Fig. 29, the client (`Job_source`) issues service requests for the software `Application` that runs on top of the `Xscale` processor. The application serves a request at the time, by requesting and releasing CPU resources by means of `Req` and `Rel` events.

Service requests are associated with two parameters: the computational `Effort` required to serve the request, and the `Deadline` imposed to the service time. Service requests are generated by the `Job_source` module in three steps. First, the `Deadline` is generated as a random variable uniformly distributed in $(0, \texttt{Max_deadline}]$. Second, a reference clock frequency f is randomly selected from the set of possible frequencies of the processor core. The reference frequency is used to obtain a feasible computational effort by multiplying the reference frequency by the given deadline: $\texttt{Effort} = f \cdot \texttt{Deadline}$. Finally, the `Request` event is generated according to a given residual-time distribution. The `Job_source` is a blocking client: no new requests are generated until the previous request has been served or its deadline has expired. To this purpose, service completion needs to be notified by the `Application` to the client.

The service time is determined by the `Application` module based on the computational effort required by the task and on the performance provided by the CPU. In par-

ticular, the Eos event is generated by the Application according to a constant residual-time distribution that uses the Effort as a task-dependent timeout value and the actual performance provided by the CPU (Perf) as a time-varying clock speed. In this way, the Eos event is automatically generated by the GSMP model as soon as the cumulative CPU performance reaches the target Effort.

Notice that both the Application and the DPM modules depend on the current state of the processor. Hence, external events are needed to notify CPU state changes to the other modules. This is done by event Change, associated with parameters Vdd and F, that represent the current operating conditions.

3) DPM Policy: The operating state of the Xscale processor core is controlled by the DPM module, which implements an application-driven DPM policy based on the knowledge of the Effort and Deadline parameters associated with the incoming service request.

The DPM decision for the nth task is taken according to the following algorithm, where indexes n and $n-1$ are used to distinguish quantities referring to current and previous tasks.

1) Compute the time required to serve request n at clock speed $F[n-1]$

$$T0 = \frac{Effort[n]}{F[n-1]}.$$

2) If $T0 >$ Deadline, increase performance.
   a) Determine the minimum values of $Vdd[n]$ and $F[n]$ compatible with the required Effort and Deadline, taking into account the time required by the processor core to switch from $Vdd[n-1]$ and $F[n-1]$.
   b) If the real-time constraints cannot be satisfied by any ($Vdd[n]$, $F[n]$) pair, the task is aborted and the CPU is kept in the Waiting state without changing its power–performance state.
   c) Else, set targetVdd $= Vdd[n]$ and targetF $= F[n]$ and issue event Tune.
3) Else, if $T0(1+$Margin$) <$ Deadline $- 30\ \mu$s, decrease performance.
   a) Determine the minimum values of $F[n]$ compatible with the required Effort and Deadline, taking into account the time required by the processor core to switch from $F[n-1]$.
   b) Determine the minimum voltage level $Vdd[n]$ compatible with $F[n]$ and with the Deadline.
   c) Set targetVdd $= Vdd[n]$ and targetF $= F[n]$ and issue event Tune.
4) Else, keep performance unchanged.
   a) Keep $F[n] = F[n-1]$.
   b) Determine the minimum voltage level $Vdd[n]$ compatible with $F[n]$ and with the Deadline.
   c) If $Vdd[n] \neq Vdd[n-1]$ set targetVdd $=$ $Vdd[n]$ and targetF $= F[n]$ and issue event Tune.

First, the service time at the current speed (denoted by $T0$) is computed and compared with the deadline imposed by the real-time constraints. If the service time is longer than the deadline, the operating speed need to be increased and the supply voltage adjusted accordingly. In this case, the DPM computes the minimum values of F and Vdd compatible with the constraints and issues a Tune event to change the power–performance state of the CPU core. If the deadline cannot be satisfied because of state transition time, the execution of the current task is aborted and the CPU is released.

If the service time is shorter than the deadline, the clock frequency can be reduced in order to save power. In this case, however, the clock adjustment overhead of 30 μs has to be taken into account. A further parameter (Margin) is used in the algorithm to decide when to slow down the processor. The meaning of the Margin will be clarified at the end of this section. Notice that when slowing down the processor, the target clock frequency is determined first, based only on the Effort and Deadline parameters, independently of the reachable supply voltage. In general, the minimum supply voltage achievable within the deadline may be higher than the minimum supply voltage required to sustain the target clock frequency because of the limited slope of the supply voltage.

Finally, if neither condition 2 nor condition 3 is satisfied, the clock frequency is kept unchanged, while possibly adjusting the supply voltage if it was above the minimum value required to sustain the current speed.

In practice, the algorithm implements a deterministic greedy policy that takes locally optimum decisions to execute the current task with the minimum power consumption, while satisfying real-time constraints whenever possible. However, the policy does not guarantee to reach a global optimum, since DPM decisions taken for task n may impair the optimality of the execution of task $n+1$. In particular, the real-time constraints associated with the tasks of our workload are satisfiable by construction, so that all deadlines could be met by operating the Xscale processor at its maximum speed. However, once the clock frequency has been reduced to take advantage of a loose constraint, the time required to adjust clock frequency and supply voltage may lead to the violation of subsequent tighter deadlines.

The Margin parameter used in the DPM policy provides a way for controlling the tradeoff between the power consumption and the probability of deadline violations. When Margin $= 0$, local DPM decisions aim at achieving the minimum power consumption for the current task, taking advantage of any opportunity for slowing down the processor regardless of the consequences for subsequent tasks. When Margin > 0, the processor speed is reduced only if there is a sizable advantage in terms of power that compensates the possible quality loss. For Margin $= \infty$, slowdown conditions are never satisfied, so that the processor works always at maximum speed, meeting all real-time constraints at the cost of a maximum power consumption.

4) Simulation Results: We performed two sets of simulations for investigating the effects of workload and DPM

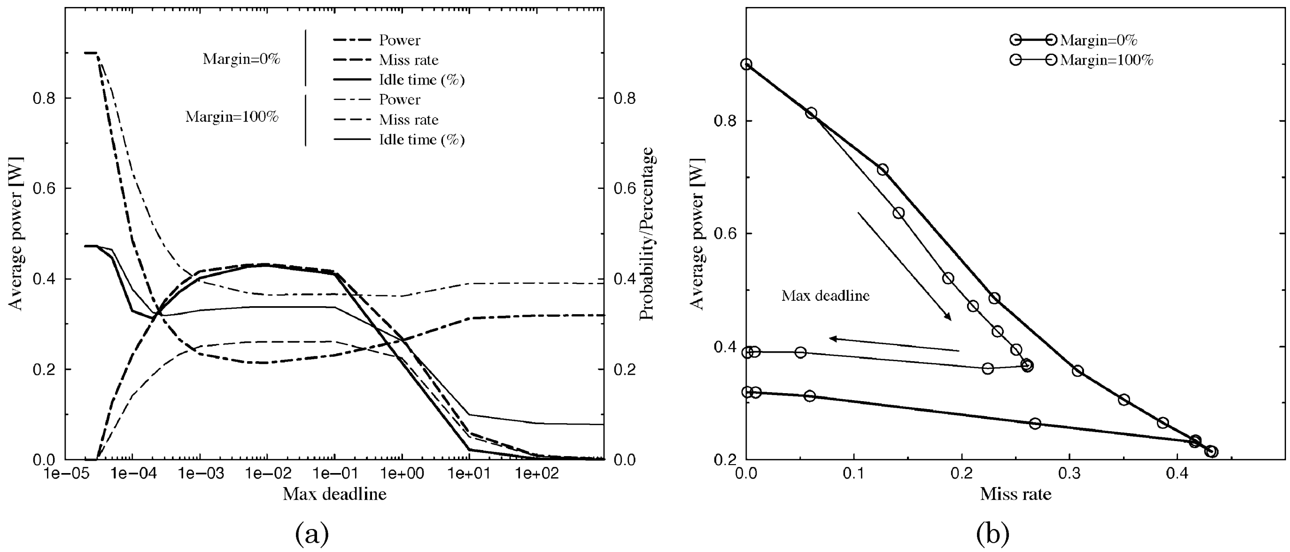


Fig. 30. Simulation results obtained for the Xscale processor under different workload conditions, obtained by changing the `Max deadline` parameter. (a) Average power, deadline miss probability and idleness, as functions of the maximum deadline of the incoming tasks. (b) Power versus miss rate tradeoff. Results are plotted for two different values of the slowdown margin used by the DPM policy.

parameters on the tradeoff between power consumption and quality of service. Results are reported in terms of average power consumption (expressed in watts), miss rate (that is the probability of violating a real-time constraint), and percentage idle time (that is the percentage of time spent by the processor core in the `Waiting` state). For each point estimate we used a workload of 10 000 tasks, randomly generated as discussed in Section V-D2.

Fig. 30(a) shows the results obtained by performing a parametric sweep on the maximum deadline for two different values of the DPM `Margin`. When the `Max_deadline` is lower than 30 μs, which is the time required to adjust the clock frequency, the operating state of the processor can never be changed. As a consequence, the power consumption is maximum (900 mW), the miss rate is null and the idleness is around 50%. The minimum power consumption is reached for `Max_deadline` around 10 ms, where the miss rate is maximum. In particular, for `Margin` $= 0$, the minimum power consumption is 200 mW, and the maximum miss rate is almost 50%.

Surprisingly enough, larger deadlines lead to higher power values. This can be explained by looking at the miss rate, which reduces to zero when `Max_deadline` is above 100 s. Since the power state of the processor is kept unchanged in case of constraint violations, each deadline miss contributes to power savings.

The same results are plotted in Fig. 30(b) as tradeoff points on a power-versus-miss-rate plane. Notice that depending on the nature of the workload, the best tradeoff is provided either by `Margin` $= 0\%$ or by `Margin` $= 100\%$.

The effects of the `Margin` used by the DPM are shown in Fig. 31(a) for different types of workloads. In summary, the higher the `Margin` the lower the power savings and miss rate and the higher the idleness. The effect of the DPM `Margin` is stronger for workloads with higher values of `Max_deadline`, because of the greater DPM opportunities.

The Pareto curves obtained by changing the DPM `Margin` are plotted in Fig. 31(b).

VI. Conclusion

In this paper, we tried to characterize and classify several techniques that fall under the name of DPM for electronic systems. The importance of DPM is well recognized, as it has been shown to be the most effective means for curbing energy dissipation. Nevertheless, the lack of general theory of DPM has hampered the evaluation and comparison of different DPM strategies and policies for their potential effectiveness.

This paper attempts to cover this void by providing a general model for power management that is widely applicable. This model is based on the use of DESs for representing system components, workloads, and controllers. In particular, the structure of these DESs is specified in terms of physical states (representing operation modes) and events (triggering state transitions), while system behavior is specified in terms of next-event and next-state functions. In this work, we use properties of GSMPs to cope with the nondeterminism of these functions.

We have shown how to build a modeling framework, with a general denotational model, supporting composition, and abstraction. The model has a rigorous execution semantics that enables event-driven simulation. The objective of the framework is to create a simulator, which is general enough to analyze different classes of DPM methods, yet based on a single and sound mathematical model. The simulator is built on top of MathWork's Simulink by using templates to execute GSMP models. These templates can be specialized to capture dynamic power-managed systems of practical interest.

We have used the simulator to evaluate and optimize system parameters and power management policies. We have analyzed in details several cases studies, including the Intel Xscale processor architecture, a multitasking

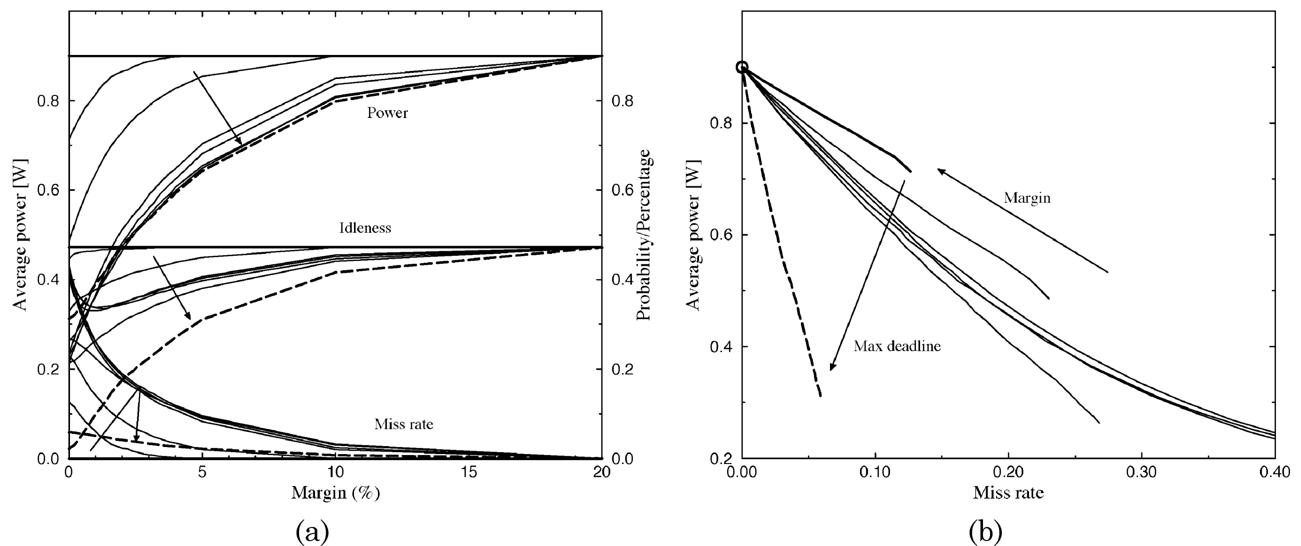


Fig. 31. Simulation results obtained for the Xscale processor with different DPM policies, obtained by changing the `Margin` parameter. (a) Average power, deadline miss probability and idleness, as functions of the slowdown margin. (b) Power versus miss rate tradeoff. Results are plotted for different values of the `Max deadline` parameter, ranging from 3 μs to 10 s.

real-time system and a sensor network. We have been able to show how the simulator can capture the essential features of power-managed systems and be used for their effective design.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for comments and suggestions.

REFERENCES

[1] R. Krashinsky and H. Balakrishnan, "Minimizing energy for wireless web access with bounded slowdown," in *Proc. Annu. Int. Conf. Mobile Computing and Networking*, 2002, pp. 119–130.
[2] V. Raghunathan, S. Ganeriwal, C. Schurgers, and M. Srivastava, "E^2WFQ: an energy efficient fair scheduling policy for wireless systems," in *Proc. Int. Symp. Low Power Electronics and Design*, 2002, pp. 12–14.
[3] D. Bertozzi, L. Benini, and B. Riccò, "Power aware network interface management for streaming multimedia," in *Proc. IEEE Wireless Communications and Networking Conf.*, vol. 2, 2002, pp. 926–930.
[4] W. Cynara and D. Bertsekas, "Distributed power control algorithms for wireless networks," *IEEE Trans. Veh. Technol.*, vol. 50, pp. 504–514, Mar. 2001.
[5] A. Goldsmith and S. Wicker, "Design challenges for energy-constrained ad hoc wireless networks," *IEEE Wireless Commun.*, vol. 9, pp. 8–27, Aug. 2002.
[6] N. Persone and V. Grassi, "Performance analysis of caching and prefetching strategies for palmtop-based navigational tools," *IEEE Trans. Intell. Transport. Syst.*, vol. 4, pp. 23–34, Mar. 2003.
[7] P. Kumar, "New technological vistas for systems and control: the example of wireless networks," *IEEE Control Syst. Mag.*, vol. 21, pp. 24–37, Feb. 2001.
[8] R. Bruno, M. Conti, and E. Gregori, "Optimization of efficiency and energy consumption in p-persistent CSMA-based wireless LANs," *IEEE Trans. Mobile Comput.*, vol. 1, pp. 10–31, Jan.–Mar. 2002.
[9] R. Badra and B. Daneshrad, "Adaptive link layer strategies for asymmetric high-speed wireless communications," *IEEE Trans. Wireless Commun.*, vol. 1, pp. 429–438, July 2002.
[10] I. Kim and H. Kim, "An optimum power management scheme for wireless video service in CDMA systems," *IEEE Trans. Wireless Commun.*, vol. 2, pp. 81–91, Jan. 2003.
[11] C. Chiasserini and R. Rao, "Improving energy saving in wireless systems by using dynamic power management," *IEEE Trans. Wireless Commun.*, vol. 2, pp. 1090–1100, Sept. 2003.
[12] N. Bambos, "Toward power-sensitive network architectures in wireless communications: Concepts, issues, and design aspects," *IEEE Pers. Commun.*, vol. 5, pp. 50–59, June 1998.
[13] T. ElBatt and A. Ephremides, "Joint scheduling and power control for wireless ad hoc networks," *IEEE Trans. Wireless Commun.*, vol. 3, pp. 74–85, Jan. 2004.
[14] A. Ephremides, "Energy concerns in wireless networks," *IEEE Wireless Commun.*, vol. 9, pp. 48–59, Aug. 2002.
[15] K. Choi, K. Soma, and M. Pedram, "Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times," in *Proc. Design Automation and Test in Eur. Conf.*, 2004, pp. 4–9.
[16] L. Feeney and M. Nilsson, "Investigating the energy consumption of a wireless network interface in an ad-hoc networking environment," in *Proc. Conf. IEEE Communications Soc.*, vol. 3, 2001, pp. 1548–1557.
[17] A. Acquaviva, E. Lattanzi, and A. Bogliolo, "Power-aware network swapping for wireless palmtop PCs," in *Proc. Design Automation and Test in Eur. Conf.*, 2004, pp. 858–863.
[18] M. Miller and N. Vaidya, "Minimizing energy consumption in sensor networks using a wakeup radio," in *Proc. Wireless Communications and Networking Conf.*, 2004, pp. 120–125.
[19] C. Chong and S. Kumar, "Sensor networks: evolution, opportunities, and challenges," *Proc. IEEE*, vol. 91, pp. 1247–1256, Aug. 2003.
[20] A. Chandrakasan and R. Brodersen, *Low-Power CMOS Design*. Piscataway, NJ: IEEE, 1998.
[21] M. Pedram and Y. Rabaey, *Power Aware Design Methodologies*. Dordrecht, The Netherlands: Kluwer, 2002.
[22] R. Viswhanath, V. Wakharkar, A. Watwe, and V. Lebonheur. (2000) Thermal performance challenges from silicon to systems. *Intel Technol. J.* [Online]. Available: http://www.intel.com/technology/itj/q32000/articles/art_4.htm
[23] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE Trans. VLSI Syst.*, vol. 8, pp. 299–316, June 2000.
[24] D. Ditzel, "Transmeta's Crusoe: cool chips for mobile computing," in *Proc. Hot Chips Symp.*, 2000.
[25] L. Clark, E. Hoffman, J. Miller, M. Biyani, L. Liao, S. Strazdus, and M. Morrow, "An embedded 32-b microprocessor core for low-power and high-performance applications," *IEEE J. Solid-State Circuits*, vol. 36, pp. 1599–1608, Nov. 2001.
[26] B. P. Zeigler, "DEVS representation of dynamical systems: event-based intelligent control," *Proc. IEEE*, vol. 77, pp. 72–80, Jan. 1989.
[27] P. W. Glynn, "A GSMP formalism for discrete event systems," *Proc. IEEE*, vol. 77, pp. 14–23, Jan. 1989.
[28] Simulink, M. Inc.. (2003). [Online]. Available: http://www.mathworks.com/products/simulink

[29] R. Ho, K. Mai, and M. Horowitz, "The future of wires," *Proc. IEEE*, vol. 89, pp. 490–504, Apr. 2001.

[30] T. Theis, "The future of interconnection technology," *IBM J. Res. Develop.*, vol. 44, pp. 42–53, 2000.

[31] T. Karnik, S. Borkar, and V. De, "Sub-90 nm technologies—Challenges and opportunities for CAD," in *Proc. IEEE Int. Conf. Computer-Aided Design*, 2002, pp. 203–206.

[32] P. Zuchowski, C. Reynolds, R. Grupp, S. Davis, B. Cremen, and B. Troxel, "A hybrid ASIC and FPGA architecture," in *Proc. IEEE Int. Conf. Computer-Aided Design*, 2002, pp. 187–194.

[33] H. Zhang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous, and J. Rabaey, "A 1-V heterogeneous reconfigurable DSP IC for wireless baseband digital signal processing," *IEEE J. Solid-State Circuits*, vol. 35, pp. 1697–1704, Nov. 2000.

[34] T. Martin and D. Sewiorek, "Nonideal battery and main memory effects on CPU speed-setting for low power," *IEEE Trans. VLSI Syst.*, vol. 9, pp. 29–34, Feb. 2001.

[35] L. Benini and G. De Micheli, "Networks on chip: a new SoC paradigm," *IEEE Computer*, vol. 35, pp. 70–78, Jan. 2002.

[36] T. Burd, T. Pering, A. Stratakos, and R. Brodersen, "A dynamic voltage scaled microprocessor system," *IEEE J. Solid-State Circuits*, vol. 35, pp. 1571–1580, Nov. 2000.

[37] L. Clark, E. Hoffman, J. Miller, M. Biyani, L. Luyun, S. Strazdus, M. Morrow, K. Velarde, and M. A. Yarch, "An embedded 32-b microprocessor core for low-power and high-performance applications," *IEEE J. Solid-State Circuits*, vol. 36, pp. 1599–1608, Nov. 2001.

[38] V. Raghunathan, C. Schurgers, S. Park, and M. Srivastava, "Energy-aware wireless microsensor networks," *IEEE Signal Processing Mag.*, vol. 19, pp. 40–50, Mar. 2002.

[39] R. Sinha, V. Liang, C. Paredis, and P. Khosla, "Modeling and simulation methods for design of engineering systems," *J. Comput. Inform. Sci. Eng.*, vol. 1, pp. 84–91, 2001.

[40] D. Estrin, D. Cuyller, K. Pister, and G. Sukhatme, "Connecting the physical world with pervasive networks," *IEEE Pervasive Comput.*, vol. 1, pp. 59–69, Jan.–Mar. 2002.

[41] R. Min *et al.*, "Energy-centric enabling technologies for wireless sensor networks," *IEEE Wireless Comput.*, vol. 9, pp. 28–39, Aug. 2002.

[42] J. Rabaey, "Wireless beyond the third generation. Facing the energy challenge," in *Proc. ACM Int. Symp. Low Power Electronics and Design*, 2001, pp. 1–3.

[43] L. Benini, G. Castelli, A. Macii, E. Macii, M. Poncino, and R. Scarsi, "Discrete-time battery models for system-level low-power design," *IEEE Trans. VLSI Syst.*, vol. 9, pp. 630–640, Oct. 2001.

[44] C. Chiasserini and R. Rao, "Energy efficient battery management," *IEEE J. Select. Areas Commun.*, vol. 19, pp. 1235–1245, July 2001.

[45] W. Kim, D. Shin, H. Yun, J. Kim, and S.-L. Min, "Performance comparison of dynamic voltage scaling algorithms for hard real-time systems," in *Proc. IEEE Real-Time and Embedded Technology and Applications Symp.*, 2002, pp. 219–225.

[46] F. Gruian, "Energy-centric scheduling for real-time systems," Ph.D. dissertation, Lund Univ., Lund, Sweden, 2002.

[47] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: exploiting generational behavior to reduce cache leakage power," in *Proc. ACM/IEEE Int. Symp. Computer Architecture*, 2001, pp. 240–251.

[48] W. Zhang, J. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. Irwin, "Compiler-directed instruction cache leakage optimization," in *Proc. Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2002, pp. 215–224.

[49] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. Irwin, "Hardware and software techniques for controlling DRAM power modes," *IEEE Trans. Comput.*, vol. 50, pp. 1154–1173, Nov. 2001.

[50] A. Law and W. Kelton, *Simulation Modeling and Analysis—Third Edition*. New York: McGraw-Hill, 1999.

[51] *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*, J. Banks, Ed., Wiley-Interscience, New York, 1998.

[52] F. Nilsen, "GMSim: a tool for compositional GSMP modeling," in *Proc. Winter Simulation Conf.*, 1998, pp. 555–562.

[53] L. Benini, A. Bogliolo, G. Paleologo, and G. De Micheli, "Policy optimization for dynamic power management," *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 813–833, June 1999.

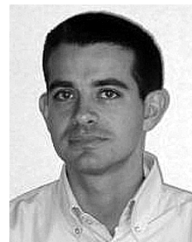
[54] V. Nicola, P. Shahabuddin, and M. Nakayama, "Techniques for fast simulation of models of highly dependable systems," *IEEE Trans. Rel.*, vol. 50, pp. 246–264, Sept. 2001.

[55] Stateflow, M. Inc.. (2003). [Online]. Available: http://www.mathworks.com/products/stateflow

[56] T. Šimunić, S. Boyd, and P. Glynn, "Managing power consumption in networks on chips," *IEEE Trans. VLSI Syst.*, vol. 12, pp. 96–107, Jan. 2004.

Alessandro Bogliolo received the Laurea degree in electrical engineering and the Ph.D. degree in electrical engineering and computer science from the University of Bologna in 1992 and 1998, respectively.

From 1992 to 1999, he was with the Department of Electronics, Computer Science and Systems (DEIS), University of Bologna. In 1995 and 1996, he was visiting the Computer Systems Laboratory (CSL), Stanford University, Stanford, CA. From 1999 to 2002, he was Assistant Professor with the Department of Engineering of the University of Ferrara. In 2002, he joined the University of Urbino, Urbino, Italy, as Associate Professor. He is currently Director of the Information Science and Technology Institute (ISTI), University of Urbino. His research interests include embedded low-power systems, dynamic power management, signal integrity, and bioinformatics.



Luca Benini received the B.S. degree (*summa cum laude*) in electrical engineering from the University of Bologna in 1991 and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1994 and 1997, respectively.

He is an Associate Professor in the Department of Electronics and Computer Science, University of Bologna, Bologna, Italy. He also holds Visiting Researcher positions at Stanford University and at the Hewlett-Packard Laboratories, Palo Alto, CA. He published more than 150 papers in international conferences and journals. He is coauthor of the book *Dynamic Power Management, Design Techniques and CAD Tools* (Boston, MA: Kluwer, 1998). His research interests are in all aspects of computer-aided design of digital circuits, with special emphasis on low-power applications, and in the design of portable systems.

Dr. Benini is a Member of the technical program committee for several technical conferences, including the Design Automation Conference, the International Symposium on Low Power Design, and the International Symposium on Hardware–Software Codesign. He is the Program Chair for the IEEE Design Automation and Testing in Europe Conference in 2005.

Emanuele Lattanzi received the Laurea degree from the University of Urbino in 2001. He is currently working toward the Ph.D. degree at the Information Science and Technology Institute (ISTI), University of Urbino.

In 2003, he was with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, working as a Visiting Scholar with Prof. V. Narayanan. His research interests are in the area of wireless embedded systems, with emphasis on power and performance analysis and optimization.

Giovanni De Micheli (Fellow, IEEE) graduated in nuclear engineering from the Polytechnic of Milan in 1979 and received the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California, Berkeley, in 1980 and 1983, respectively.

He is currently Professor of Electrical Engineering and, by courtesy, of Computer Science at Stanford University, Stanford, CA. He is, or has been, Member of the technical advisory boards of several companies, including Magma Design Automation, Coware, Aplus Design Technologies, Ambit Design Systems and STMicroelectronics. He is the author of *Synthesis and Optimization of Digital Circuits* (New York: McGraw-Hill, 1994) and coauthor and/or coeditor of five other books and over 270 technical articles. His research interests include several aspects of design technologies for integrated circuits and systems, with particular emphasis on synthesis, system-level design, hardware/software codesign and low-power design.

Dr. De Micheli is a Fellow of the Association for Computing Machinery. He is the recipient of the 2003 IEEE Emanuel Piore Award for contributions to computer-aided synthesis of digital systems. He received the Golden Jubilee Medal for outstanding contributions to the IEEE Circuits and Systems (CAS) Society in 2000. He received the 1987 D. Pederson Award for the best paper on the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN and two Best Paper Awards at the Design Automation Conference in 1983 and in 1993. He is Past President of the IEEE CAS Society. He was Editor-in-Chief of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN from 1987 to 2001. He was the Program Chair and General Chair of the Design Automation Conference (DAC) in 1996–1997 and 2000, respectively.