Energy-Aware Runtime Scheduling for Embedded-Multiprocessor SOCs

Peng Yang, Chun Wong, Paul Marchal, Francky Catthoor, Dirk Desmet, Diederik Verkest, and Rudy Lauwereins
IMEC

This task-scheduling method for embedded systems combines the low runtime complexity of a design-time scheduling phase with the flexibility of a runtime scheduling phase.

With the rapid evolution of submicron-process technology, manufacturers are integrating increasing numbers of components on one chip. A heterogeneous system on a chip (SOC) such as that shown in Figure 1 might include one or more programmable components—general-purpose processor cores, digital signal processor cores, or application-specific intellectual property cores—as well as an analog front end, on-chip memory, I/O devices, and other application-specific ICs. Unfortunately, design technologies have fallen behind advances in processing technology, especially in the context of complex (possibly data-dominated) and very dynamic (partly nondeterministic) applications. SOC designers need a consistent system design technology that can cope with such characteristics and with ever-shortening time-to-market requirements. This technology should efficiently map these dynamic applications to the target realization while meeting all real-time and other constraints.

Contemporary software (and hardware) design practices for this target class can be described only as ad hoc. The design trajectory starts by identifying the global specification entities, called tasks or processes, that functionally belong together. The second step is manual hardware-software partitioning. Because the software and hardware tasks are implemented separately, system integration is inevitable. This manual step embeds the system software and synthesizes the interface hardware that closes the gap between the software and hardware components.

The main goal of system-software embedding is to encapsulate the concurrent tasks in a control shell that handles task scheduling (software scheduling in the restricted sense) and intertask communication. Task scheduling is an error-prone process that requires computer assistance to consider the many interactions among constraints. Current ad hoc industrial design practices for reactive real-time systems are not very retargetable from one design to another, and adapting a design's behavior to a changing environment is difficult. Designers have used real-time operating systems or kernels to solve some scheduling problems. With either method, designers satisfy timing constraints by tuning the code to a specific processor and a particular I/O configuration. This results in poor modularity and limited retargetability, severely discouraging exploitation of the codesign space, even if the program is written in a high-level language.

Hence, a need exists for a systematic methodology and system-level tool support, including concurrent-task management. The lack of concurrent-task management often results in an iterative and error-prone design cycle. At the top of the design process should be a unified specification model capable of representing system-level abstractions such as process concurrency, interprocess communication and synchronization, and real-time constraints. Real-time Java extensions are a possible example of such a model, but many designers use C++ with an underlying concurrent-task-oriented class library. The main problem is to close the gap between the specification of concurrent, communicating processes and the heterogeneous processor target without compromising required real-time performance and cost-effectiveness (especially in terms of energy consumption).

In another publication, we described our overall approach to the problems raised here—a global task-concurrency management system developed at IMEC (Belgium's Interuniversity Microelectronics Center).[1] Now, we go into more detail about one element—a task-scheduling method for concurrent and communicating tasks on multiple processors. This method combines design-time and runtime scheduling and exploits the cost-performance trade-off at runtime.

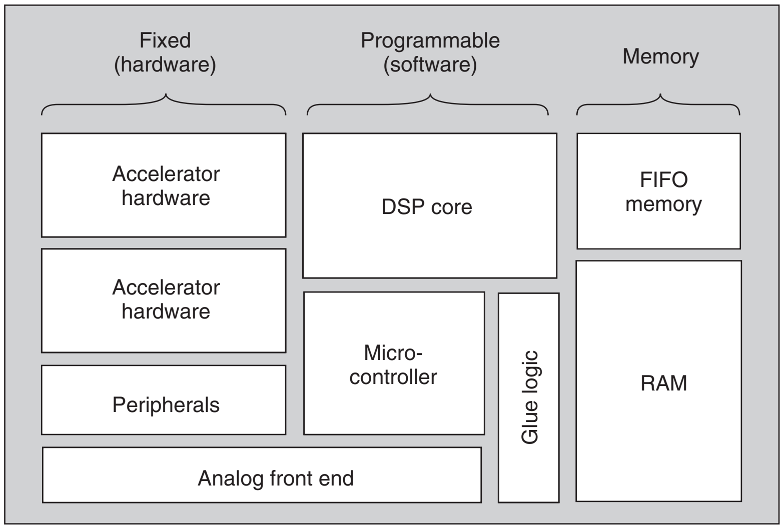


Figure 1. Typical heterogeneous SOC architecture.

Energy-aware scheduling

When one or more processors must execute a set of concurrent tasks, a predefined scheduling algorithm must be applied to decide the task execution order. For a multiprocessor system, an assignment procedure must determine which processor will execute each task. Task scheduling in a task-concurrency-management context has been investigated a great deal. Comprehensive overviews of scheduling algorithms for real-time systems are available.[2,3] These algorithms fall into two categories: dynamic and static scheduling. For multiprocessors, when the application uses a large amount of nondeterministic behavior, dynamic scheduling has the flexibility to balance the computation load at runtime. However, runtime overhead, especially computation time, may be excessive. For embedded systems, cost factors, such as energy, also must be considered. Dynamic voltage scheduling significantly reduces system energy consumption by decreasing the supply voltage. More detailed discussion of state-of-the-art scheduling algorithms is available.[4]

Manually designing concurrent real-time embedded systems—embedded software in particular—is difficult because of complex consumer-producer relationships, various timing constraints, the specification's nondeterminism, and, sometimes, the software's tight interaction with underlying hardware. Our approach addresses these problems by preordering the concurrent behavior as much as possible at design time to minimize runtime overhead. At the same time, the energy-aware runtime scheduler tries to minimize costs such as energy consumption.

The example in Figure 2 (next page) illustrates the basic idea of our method. (The "Terminology" box defines its main concepts.) In Figure 2a, two threads exist originally. The first two Pareto curves, representing the cost-performance trade-off for threads 1 and 2, are computed in the design-time scheduling phase. The diamonds indicate the operation points for a total of 200 system cycles. When thread 3 enters the system (from a Java applet, for example), the runtime scheduler considers the Pareto curves of all three threads to find a new optimal distribution of operating points for the system (Figure 2b). Assuming a cost-performance trade-off can be explored at design time, this runtime scheduling can be applied to both data- and control-dominated applications.

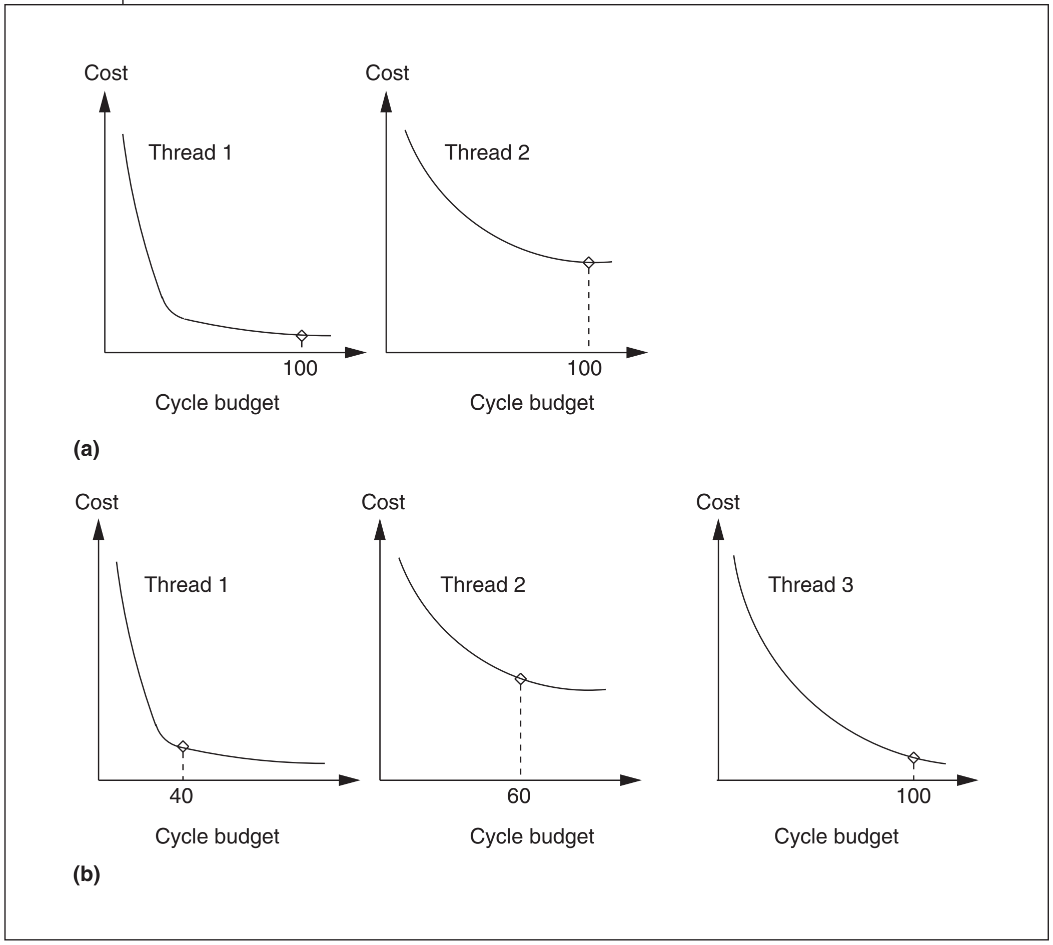


Figure 2. The optimal system solution for a three-thread system: before the third thread enters the system (a) and after the third thread enters the system (b).

As the first step in our method, designers specify an embedded system at a gray-box abstraction level in a combined multitask-graph (MTG) and control-dataflow graph (CDFG) model.[2] The specification represents concurrency, time constraints, and interaction at either an abstract or a more detailed level, depending on what the designer needs to make good exploration decisions later.

The purpose of task-concurrency management is to determine a cost-optimal, constraint-driven scheduling, allocation, and assignment of various tasks to a set of processors. Different processors execute the same thread node at different speeds and different costs (energy consumption). These differences make it possible to explore the cost-performance trade-off at the system level.

Task-concurrency management comprises three steps. The first is concurrency extraction, which produces a set of thread frames. Each thread frame consists of many thread nodes, the basic scheduling units. Second, design-time scheduling is applied inside each thread frame at compile time, including a processor assignment decision in the case of multiple processing elements. Finally, runtime scheduling is applied to these thread frames on the given platform.

We separate task scheduling into two phases for three reasons. First, this scheme better optimizes the embedded software design. Second, it gives the entire system more runtime flexibility. Third, it reduces runtime computation complexity.

Terminology

Nondeterminism: a state in which behaviors (such as latency or execution time) can vary even with the same system input. Interrupts and events can cause nondeterminism.

Pareto curve: a set of Pareto-optimal points. Each point represents an optimal solution in at least one trade-off direction when all other directions are fixed.

Thread: a group of thread frames; an independent piece of code that performs a specific function.

Thread frame: a group of thread nodes. By definition, nondeterministic behaviors can occur only at the boundary of thread frames. The design-time scheduler works inside each thread frame, whereas the runtime scheduler treats a thread frame as an atomic scheduling unit.

Thread node: the atomic scheduling unit of our design-time scheduler; consists of control-dataflow graph (CDFG) nodes and arcs.

Figure 3 illustrates the two-phase scheduling scheme. Given a thread frame, the design-time scheduler explores different assignment and schedule possibilities. Unlike other schedulers, it gives not just one solution but a Pareto-optimal set represented by a Pareto curve. Every point in the set is better than any other solution in at least one way. That is, it consumes the least energy under a given time constraint or it finishes earliest under a given energy consumption constraint. Design-time scheduling takes place at compile time, so the design-time scheduler can exert as much computation effort as necessary—provided that it produces a better result, thus reducing the computation effort of runtime scheduling later.

The runtime scheduler works at thread frame granularity. When new thread frames come into being, the runtime scheduler tries to satisfy their time constraints and minimize system energy consumption as well. The details inside a thread frame, such as execution time or each thread node's data dependency, remain invisible to the runtime scheduler, reducing the thread frame's complexity significantly. The design-time scheduler passes only a few useful Pareto curve features to the runtime scheduler, which uses them to find a reasonable cycle budget distribution for all the running thread frames. Thus, the runtime scheduler is not a traditional dynamic scheduler because it must choose from available options in addition to scheduling them.

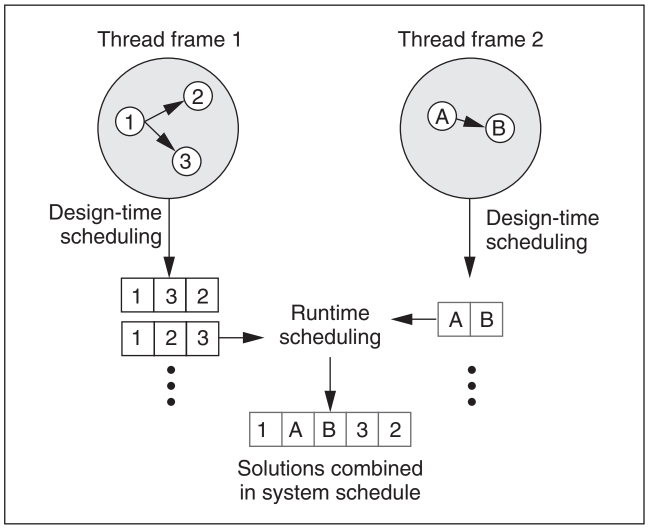
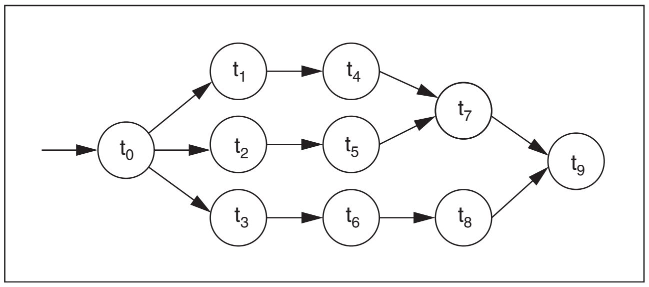


Figure 3. The two-phase scheduling method. At design time, the scheduler tries various scheduling and assignment combinations for each thread frame. Here, for example, the scheduler considers two solutions for thread frame 1—one in which node 3 is scheduled before node 2, and another in which node 3 is scheduled after node 2. At runtime, the scheduler chooses one solution for each thread frame and combines them in the system schedule.



$$\begin{tikzcd} & t_1 \arrow[r] & t_4 \arrow[dr] & & \\ \arrow[r] & t_0 \arrow[u] \arrow[r] \arrow[d] & t_2 \arrow[r] & t_5 \arrow[r] & t_7 \arrow[dr] & \\ & t_3 \arrow[r] & t_6 \arrow[r] & t_8 \arrow[r] & t_9 \end{tikzcd}$$

Figure 4. Task graph example.

Design-time scheduling phase

A thread frame's behavior can be described by task graphs such as Figure 4, in which each node represents functions to be performed and their performance requirements. Each edge represents the data dependency between two nodes. This task graph is a simplified subset of the MTG-CDFG model, and each function is a thread node in the MTG-CDFG. The task graph represents part of a voice coder and will be mapped to a dual-processor platform.[5] The two processors are almost the same, except that P1's working voltage is three times that of P2.

Table 1 shows each node's performance on the two processors in terms of execution time and energy consumption. We have normalized the execution time and energy consumption numbers because the absolute value is not important in this context. We explain the relations among working voltage, execution time, and energy consumption later. Although we do not show them here, we can add some time constraints to the task graph; for instance, task t_4 must start n time units after task t_0 ends. We scheduled the thread frame nonpreemptively because we have a priori knowledge of all the nodes involved.

Any suitable static task-scheduling algorithm for multiprocessors can be applied to

Table 1. Thread node performance.

Processor	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
Execution time (normalized)										
P1	3	10	12	13	16	13	15	30	20	15
P2	9	30	36	39	48	39	45	90	60	45
Energy consumption (normalized)										
P1	27	90	108	117	144	117	135	270	180	135
P2	3	10	12	13	16	13	15	30	20	15

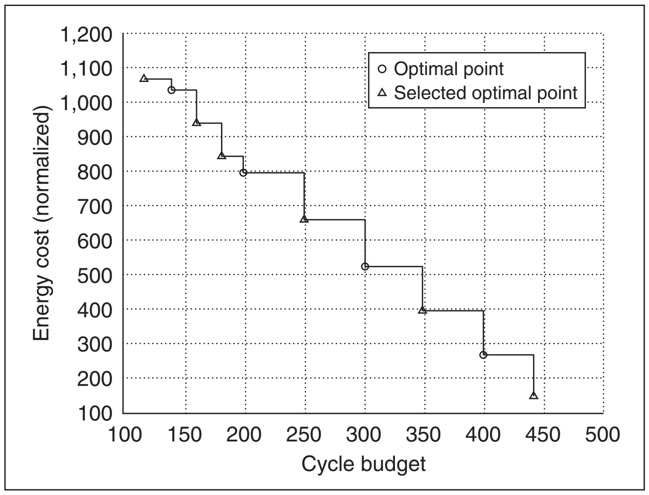


Figure 5. Pareto curve for design-time scheduling of voice coder.

such a thread frame, as long as it can produce the cost-performance trade-off as a Pareto curve. Here we use a genetic algorithm for design-time scheduling because of its speed and near-optimal solution. Figure 5 shows the Pareto curve we derived with this algorithm for the voice coder example. Among the Pareto-optimal set of points, only those chosen by the design-time scheduler as typical cases (indicated by triangles) are passed to the runtime scheduler. The more points passed, the better the runtime scheduler's results, but at the cost of greater runtime computation complexity and overhead.

Genetic algorithms maintain a pool of solutions that evolve in parallel over time. Genetic operators are applied to the solutions in the current pool to improve them. The lowest-quality solutions are then removed from the pool.[6] A cost is a variable—in our case, energy consumption—that a genetic algorithm attempts to minimize. Compared to optimal algorithms such as mixed-integer linear programming (MILP) or exhaustive search, genetic algorithms cannot promise an optimal result. In most cases, however, with a set of finely tuned parameters, genetic algorithms can give a good enough result because they can escape local optimal points and communicate information among solutions. For nontrivial-size problems, optimal algorithms usually aren't applicable because they require excessive computing time, but genetic algorithms give good results in reasonable time.

We used the Parallel Genetic Algorithm (PGA) library[7] to implement our genetic algorithm because it gave us the flexibility and simplicity to design our own algorithm. The algorithm takes a deadline as input and tries to find a schedule that consumes minimal energy within that deadline.

In a genetic algorithm, a *chromosome*, or *string*, represents every solution. Three operators cause all changes to strings: *reproduction* makes a copy of a solution, *mutation* randomly changes part of a solution's description, and *crossover* swaps portions of different solutions. Crossover gives a genetic algorithm its strength by letting different solutions share information with one another. By using these three operators repeatedly, a genetic algorithm generates children from the previous generation. Then it uses a cost function to evaluate solutions, and only solutions with good health survive. This evolution continues until a satisfactory solution is found or the iterations have exceeded a predefined number.

To solve the processor-assignment and task-scheduling problems simultaneously, the algorithm represents each thread node t_i by two genes in the string[8]—one for the thread node's allocation destination $A(t_i)$, and another for its priority number $P(t_i)$. The priority number determines the thread node execution order. To represent a valid schedule, a string must take the precedence constraints into account. Therefore, after mutation and crossover, the algorithm must repair strings that don't respect the precedence constraints because of these two operators' random behavior.

Before the evaluation step, the algorithm schedules the thread nodes according to their allocation destinations and priority numbers. A node with a higher priority number is scheduled earlier, and it starts only when its processor is free and all its precedents have finished. After scheduling, the algorithm computes the entire task graph execution time. If it exceeds the desired deadline, a penalty term is added to the cost function. This penalty should be large enough to distinguish valid and invalid solutions, but it should not be too large. Therefore, invalid solutions that contain portions consisting of good genes can still survive in the population. These good portions might transfer to a valid solution in the next generation, but this is only a possibility and is not controlled by the algorithm (this randomness partly accounts for the advantage of genetic algorithms). Even a well-designed algorithm can miss this possibility.

Runtime scheduling phase

Design-time scheduling provides a series of possible allocation and scheduling options inside a thread frame, but only the runtime scheduler decides which option is used. Each option has a thread node assignment and scheduling pattern computed by the design-time scheduler. The runtime scheduler considers computation requests from all the ready-to-run thread frames and selects an option for each thread frame so that the entire system's combined energy consumption is optimal. Working with the thread frame as its operational unit, the runtime scheduler considers the timing constraints among thread frames, such as data dependency or execution order.

In Table 2, for example, each of two thread frames has three options that were identified by the design-time scheduler. These options correspond to different cycle budget and energy cost combinations. At runtime, if the total cycle budget for the two thread frames is 100, the energy-optimal schedule is option 1 for thread frame 1 and option 3 for thread frame 2. If the cycle budget is 140, however, the optimal schedule becomes option 2 for thread frame 1 and option 3 for thread frame 2. Figure 6 depicts both cases. Complex trade-offs are involved in distributing the total execution period over the different thread frames.

Table 2. Two thread frames with options for runtime scheduling.

Trade-off	Thread frame 1			Thread frame 2		
factors	Opt. 1	Opt. 2	Opt. 3	Opt. 1	Opt. 2	Opt. 3
Cycle budget	20	60	100	40	60	80
Energy cost	110	80	50	90	60	50

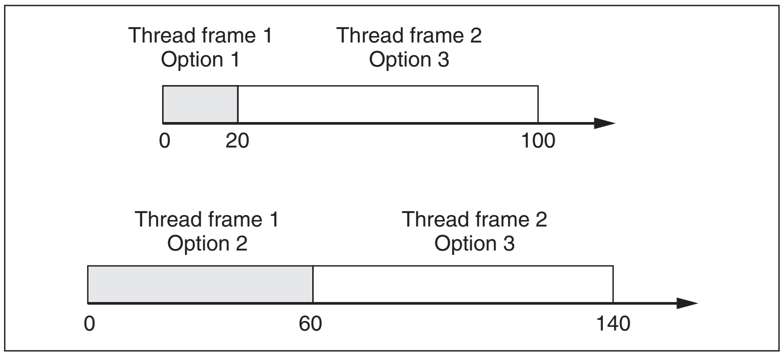


Figure 6. Runtime scheduling of two thread frames.

We used a mixed-integer linear programming algorithm to model the runtime scheduling problem. Although it is too time-consuming to be used as an online algorithm, the MILP algorithm gave us a good idea of how well our method would work. We are seeking a heuristic replacement in our ongoing research.

The MILP runtime scheduling algorithm uses the following variables:

- $\delta_{i,j} = 1$ if the jth option of thread frame i is used; $\delta_{i,j} = 0$ otherwise.
- $C_{i,j}$ is the execution time of thread frame i, option j.
- $E_{i,j}$ is the energy consumption of thread frame i, option j.

For n thread frames, our aim is to minimize all the thread frames' energy consumption. Therefore, we define the object function as

$$\text{Minimize:} \sum_{i,j} E_{i,j}\delta_{i,j}$$

For each thread frame, the runtime scheduler chooses one and only one Pareto option. For this choice, the constraints are

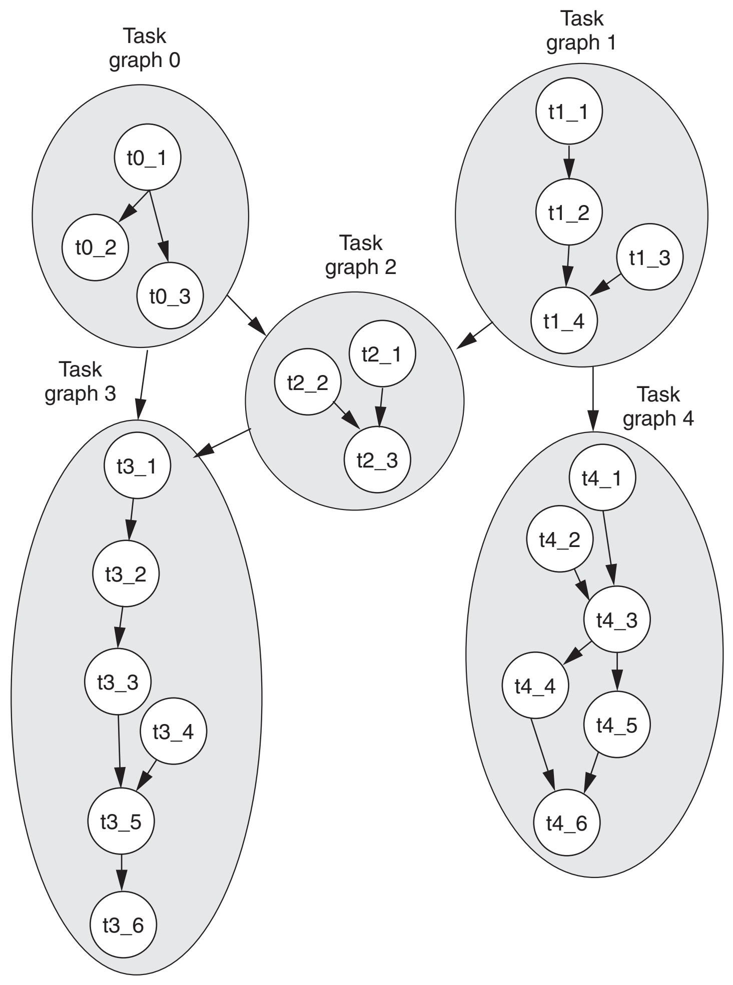


Figure 7. Five task graph examples.

$$\sum_j \delta_{i,j} = 1, \text{ for every } i \le n$$

Also, the total execution time should be less than the total available time units:

$$\sum_{i,j} C_{i,j}\delta_{i,j} \le \text{ time units}$$

So far in our work, we have focused on mutually exclusive thread frames. When one thread frame executes, it occupies all the processors and other frames cannot use them, even though some processors may be idle at the time. This is not a limitation in practice, however, because thread frames consist of many thread nodes, which can be effectively distributed over the processors to avoid nearly all potential idle time.

Experiments with randomly generated task graphs

We first applied our two-phase scheduling procedure on some randomly generated examples to test its effectiveness. We generated these pseudorandom examples with the Task Graphs for Free (TGFF) system.[9]

TGFF-generated task graphs

We generated five task graphs, each of which can be considered a thread frame containing a different number of thread nodes. These thread frames can comprise a higher-level thread, as shown in Figure 7, or they can be independent of each other.

To the runtime scheduler, the thread frame is an unsplittable block, but because of the system's dynamic behavior, a thread frame can appear or not appear (that is, be part of the system or not be part of the system) during the current iteration. This is a good simulation of dynamic mechanisms (such as dynamic creation and release) caused by the presence of events and semaphores. Dynamic behaviors are common in advanced communication or multimedia systems. However, most current scheduling methods model them poorly, and they are considered only in worst-case scenarios. In our scheduling procedure, we restrict these nondeterministic behaviors to the thread frame boundary; that is, nondeterminism can occur only at the boundary. Whenever nondeterminism triggers a thread frame, all the nodes inside that thread frame are to be executed.

Table 3 lists the basic characteristics of the TGFF-generated task graphs. It also gives the normalized execution time and energy consumption for executing the entire task graph on a single processor with a 3-V working voltage (V_h).

Processor architecture

In this and the following experiments, we assumed that the system uses two homogeneous processors running in parallel at two different voltages (V_h and V_l). This architecture differs from the dynamically variable voltage single-processor architecture and the unique-voltage multiple-processor architecture. Our architecture selection is reasonable: If only one fixed-voltage processor is present, it must be fast

Table 3. TGFF-generated task graphs.

Parameter	Task graph 0	Task graph 1	Task graph 2	Task graph 3	Task graph 4
Number of nodes	8	47	12	29	21
Number of arcs	9	59	13	35	26
Normalized execution time (V_h)	284	1,257	371	826	661
Normalized energy consumption (V_h)	2,228	12,114	3,316	8,147	6,717

enough to handle heavy load bursts, and a fast processor is typically power greedy. However, heavy load bursts occur only occasionally. At other times, all the tasks execute on that fast, power-greedy processor, although it need not be so fast. Even with modern power control techniques that shut down the fast processor during idle times, it still consumes more energy than two processors working at different voltages. Unlike the normal multiprocessor architecture of homogeneous processors, our system consists of processors that provide different cost-performance trade-offs. But because all the processors are the same type, code can easily be assigned to any of them dynamically.

The architecture we chose has several advantages over the dynamically variable voltage single-processor architecture. First, a multiprocessor can exploit an application's internal parallelism, and the trend is to integrate multiple processors on a single chip. Second, the combination of multiprocessor and discrete-level working voltage can provide a similar energy consumption savings as a processor with a continuously variable working voltage.[10] Third, our architecture avoids an additional voltage-tuning circuit and runtime voltage-swapping overhead.

In well-designed CMOS digital circuits, the dominant energy consumption term is the switching component,[11] which is

$$\begin{aligned}\text{energy per transition} &= P_{\text{total}} / f_{\text{clk}} \\ &= C_{\text{effective}} V_{\text{dd}}^2\end{aligned}$$

and the maximal frequency is

$$f_{\max} = \frac{1}{T_{\text{d}}} = \frac{\mu C_{\text{ox}}(W/L)(V_{\text{dd}} - V_{\text{Th}})^2}{C_{\text{L}} V_{\text{dd}}}$$

In our experiment, we assumed that one processor works at 1 V, and the other works at 3 V; thus, the high-voltage processor consumes roughly nine times as much energy as the low-voltage one. To simplify the experiment, we assumed the high-voltage processor works three times as fast as the low-voltage one. These assumptions don't impair our method's correctness in cases where the relations among energy, speed, and voltage scaling are different. Other reasonable assumptions were that the processor powers down automatically when it is idle, and that no context-switch overhead is considered.

Results

For each of the five task graphs, our genetic algorithm created a list of Pareto options and selected eight points from the list to represent the design-time scheduling decision for that task graph. To simulate the system's dynamic behavior, we randomly created dynamic patterns for 10 periods; the patterns determine whether or not a task graph will execute in each period. Table 4 (next page) shows the dynamic patterns.

We compared our scheduling result with several cases, assuming a period of 5,000 or 4,000 time units. First, we considered a one-voltage processor. Because a low-voltage processor cannot execute such a heavy load alone, we considered only the high-voltage case.

Next, we assumed a system with the same dual-processor architecture as ours. But unlike our dynamic working-point selection, its working point is fixed at design time. That is, each task graph can work only under one Pareto option; this point is fixed at compilation and is not movable at runtime.

We can divide this case into two subcases, depending on how we determine the static working point. In the first subcase, we divide the entire period evenly among the five task

Table 4. Randomly generated dynamic patterns of task graph execution.

Period	Task graph pattern				
0	TG1	TG3	TG4		
1	TG0	TG1	TG2	TG3	TG4
2	TG1	TG2	TG3	TG4	
3	TG0	TG1			
4	TG0	TG1	TG2		
5	TG3	TG4			
6	TG2	TG4			
7	TG0	TG1	TG4		
8	TG0	TG2	TG3		
9	TG1	TG2			

graphs. This is an unwise decision because the five task graphs differ significantly in size and computing resource requirements. For example, the system becomes unschedulable when the period is 4,000 time units because TG1, the largest task graph, cannot fit in the 800 time units assigned to it.

In the second subcase, we tune the system to the most critical load situation. In other words, we use the working points optimized for period 1 for all 10 periods. By doing that, we can guarantee that the system is still schedulable even under the heaviest load. This is exactly the way most schedulers, whether static or dynamic, behave. A real-time operating system also bases its scheduling on such worst-case analysis.

Table 5 shows our results. The second column gives the energy consumed if all thread frames execute on one high-working-voltage processor. The following columns list the percentage improvements over the one-processor architecture for the evenly divided, critically tuned, and dynamically selected cases.

These results show that our two-processor architecture significantly reduces energy consumption compared with the one-processor platform. When the period is 5,000 time units, even a simple schedule that divides the available computing resources evenly among the task graphs can reduce energy consumption 48%. Our two-phase scheduling method defeats the other two methods with a total energy savings of up to 72%. Notice that the more the worst case differs from the average case, the more energy we can save. And that is typically true for dynamic multimedia applications such as MPEG-4.

ADSL experiments

To further assess our approach, we applied it to an industrial-strength application, an asynchronous digital subscriber line modem.

System architecture

Figure 8 shows the ADSL modem's system architecture. The design consists of hardware components and embedded software with real-

Table 5. Results on randomly generated task graphs.

		Improvement over one-processor architecture (%)				
		Period = 5,000 time units			Period = 4,000 time units	
Period	One-processor (V_h) energy consumption (normalized)	Evenly divided	Critically tuned	Dynamically selected	Critically tuned	Dynamically selected
0	26,978	42	53	64	46	54
1	32,522	49	52	52	43	43
2	30,294	46	54	57	44	47
3	14,342	41	44	89	44	79
4	17,658	49	47	80	41	68
5	14,864	49	57	89	45	82
6	10,033	61	68	89	44	89
7	21,059	45	52	73	47	62
8	13,691	62	48	89	34	81
9	15,430	43	51	89	43	77
Total	196,871	48	52	72	44	63

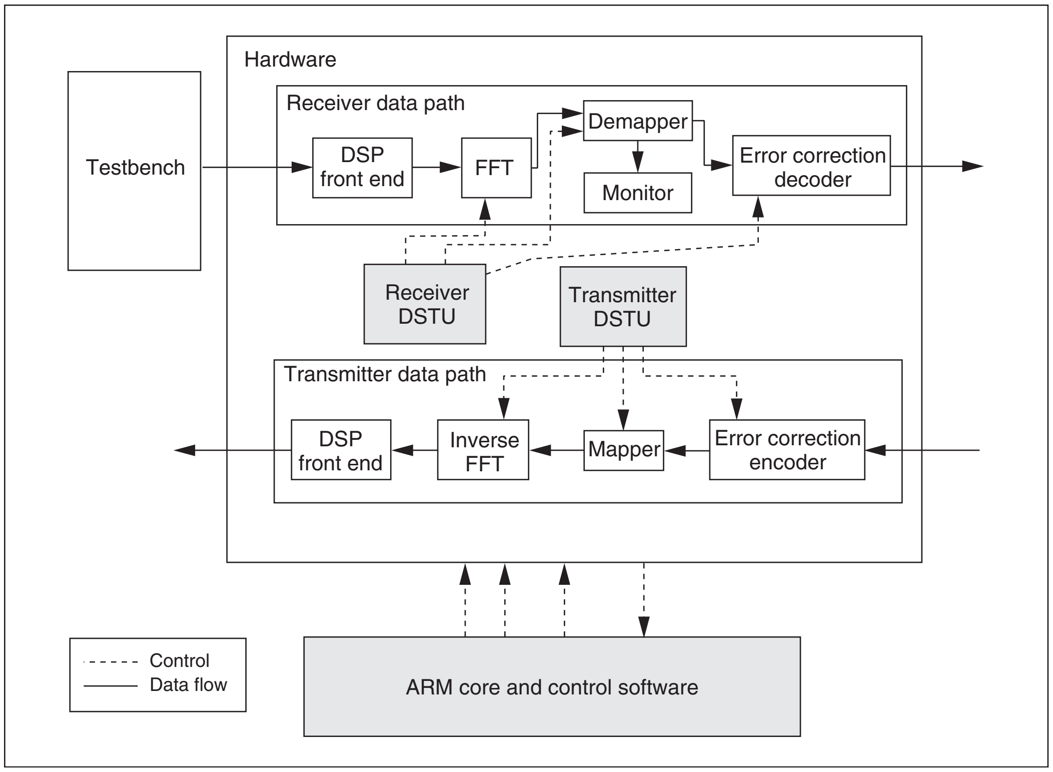


Figure 8. ADSL modem schematic.

time constraints. The digital hardware includes two parallel data paths for the receiver and transmitter. The receiver data path consists of a front end, a fast Fourier transformer, a quadrature amplitude modulation (QAM) demapper, an error and noise monitor, and an error correction decoder. The transmitter data path has a similar structure. Both data paths have hardware timing controllers (DMT symbol timing units). These DSTUs activate the processors at the correct moments to appropriately process discrete multitone (DMT) symbols.

Next to the hardware components, an Advanced RISC (reduced instruction-set computing) Machine (ARM) core processor runs the embedded software that programs and configures the hardware. The software's control part configures the hardware to execute the initialization sequence, and its algorithmic part executes DSP functionality not implemented in hardware. The control part is a reactive system, reacting to events generated by the monitor and other hardware modules, and meeting real-time constraints imposed by the ADSL standard.

In our experiment, we focused on the system consisting of the DSTU and software modules (the shaded modules in Figure 8). The DSTU controllers generate two thread nodes (one from the transmitter and one from the receiver) every symbol period (230 µs), and the applied deadline equals the period. The nodes from one or more periods comprise one thread frame. The software module generates another thread frame dynamically when the corresponding event has been triggered. The deadline for the software thread frame is derived from the ADSL standard. The software is executed serially; that is, no new software thread frame is generated while another is being processed.

Assumptions

The DSTU thread nodes work at a rigid 230-µs symbol period, and each node requires 64 µs of computation time on a 10-MHz processor. The software thread is a sequence of sporadic tasks, which are released by the hardware and must be completed by their deadlines. The deadlines and execution times are listed in Table 6 (next page). We extracted the deadlines from the ADSL standard's initialization sequence.

Table 6. Software task deadlines and execution times.

Task	Deadline (symbol)	Execution time at 10 MHz (ms)
4	128	3
5	96	3
6	128	3
7	128	21
8	768	240
9	256	9
10	128	60
11	132	12
12	16	3
14	64	21
15	16	3
16	2,048	864
17	64	18
18	576	285

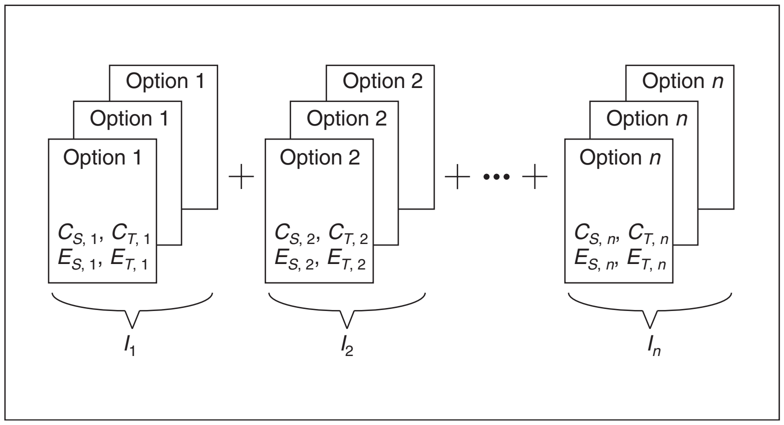


Figure 9. Runtime scheduling of the ADSL modem.

Runtime scheduling

We used two threads in this experiment; one consists of the two DSTUs, and the other is the software controller. For the software thread, we dynamically generated a thread frame that has only one thread node. For the timer thread, we generated two thread nodes every symbol period, and a timer thread frame can cover the thread nodes from one, two, or even more symbol periods. The more symbol periods a timer thread frame covers, the lower the processor utility ratio, owing to the increasing intra-thread-frame idle time.

There is another difference from the previous experiment. The software thread's execution time and deadline, varying from tens to hundreds of symbol periods, are far longer than the timer thread frame's time granularity, which is only a few symbol periods. Therefore, one software thread frame must be scheduled simultaneously with many timer thread frames, and they intersect the software thread frame into many pieces. Accordingly, we changed the runtime scheduling method a little.

Figure 9 illustrates the method we used for the ADSL modem. To formalize the problem, suppose we have n design-time scheduling options for one timer thread frame that covers k symbol periods. For each option i, $C_{T,i}$ and $C_{S,i}$ ($i = 1, 2, \ldots, n$) represent the time that the system can use to execute the timer and software thread, respectively, in the k symbol periods. $E_{T,i}$ and $E_{S,i}$ ($i = 1, 2, \ldots, n$) represent the energy consumption for the timer and software in that interval. The runtime scheduler works at the granularity of k symbols. Also suppose that the coming software thread node's execution time is C and the deadline is D (in symbols). We have n possible choices for a timer thread frame, and we let l_i ($i = 1, 2, \ldots, n$) represent the number of timer thread frames for each scheduling choice. Therefore, to find a feasible schedule, we must find a set of l_is that can provide enough execution time for that software thread node before the deadline. To find an optimal energy-cost schedule, we must choose the one with minimal energy consumption among these feasible schedules.

We can restate the runtime scheduling method as an MILP problem:

$$\sum_{i=1}^{n} l_i \cdot C_{S,i} \geq C \tag{1}$$

$$\sum_{i=1}^{n} l_i \leq \frac{D}{k} \tag{2}$$

$$\text{Minimize: } \sum_{i=1}^{n} l_i \cdot \left(E_{S,i} + E_{T,i}\right) \tag{3}$$

Equation 1 is the constraint on software execution time, equation 2 makes sure the software finishes executing before the deadline, and equation 3 is the optimizing objective function. By solving that MILP problem, we obtain an energy-optimal, deadline-satisfying schedule.

Results and analysis

Our experiment considered five cases. Case 1 was scheduled at the thread-node level; thus, it exhibits no intra-thread-frame idle time. Cases 2, 3, 4, and 5 were scheduled at the thread-frame level, and they had one, two, three, and four symbols, respectively, in one timer thread frame. For a fixed number of thread nodes, more thread nodes in one thread frame means fewer thread frames and less runtime effort for the runtime scheduler—at the cost of losing some optimization possibilities. This is obvious in Table 7, which shows the normalized energy costs for the five cases.

Table 7. Energy costs of different cases at the same working voltages.

Case	Energy cost (normalized)	Energy savings (%)
1	31.2	39
2	34.6	32
3	34.6	32
4	39.1	24
5	41.1	22

While the timer thread frame's granularity increases, the energy consumption usually increases too. If only one processor is working at voltage V_h, the normalized energy consumption is 51.2. So, compared with the one-processor architecture, our architecture achieves an energy savings of about 20% even at the largest thread frame granularity in case 5. We cannot achieve the same high energy reduction ratio as in the previous experiment because of the highly restricted deadlines, especially for the big tasks 8, 16, and 18. The energy-saving percentage also varies with the system load. The heavier the load, the less the savings percentage because the high-voltage processor does more work. If all the tasks are assigned to the high-voltage processor, the savings are reduced to those of the one-processor case.

The results of the ADSL experiment are promising. First, we obtained energy savings of 20% to 40%. Second, we changed the granularity of a thread frame and demonstrated that a trade-off exists between dynamic-scheduling overhead and overall scheduling optimality.

Our two-step task-scheduling method increases design flexibility and reduces design time for multiprocessor SOCs, while minimizing global system energy costs. Our future work includes applying the method to additional realistic applications and integrating it into operating systems. ■

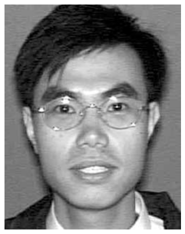
References

1. A. Prayati et al., "Task Concurrency Management Experiment for Power-Efficient Speedup of Embedded MPEG4 IM1 Player," *Proc. Int'l Conf. Parallel Processing*, IEEE CS Press, Los Alamitos, Calif., 2000, pp. 453-460.
2. F. Thoen and F. Catthoor, *Modeling, Verification and Exploration of Task-Level Concurrency in Real-Time Embedded Systems*, Kluwer Academic, Boston, 1999.
3. K. Ramamritham and J.A. Stankovic, "Scheduling Algorithms and Operation Systems Support for Real-Time Systems," *Proc. IEEE*, vol. 82, no. 1, Jan. 1994, pp. 55-67.
4. C. Wong et al., "Task Concurrency Management Methodology to Schedule the MPEG4 IM1 Player on a Highly Parallel Processor Platform," *Proc. Int'l Workshop Hardware/Software Codesign*, IEEE CS Press, Los Alamitos, Calif., 2001, pp. 170-175.
5. C. Lee, M. Potkonjak, and W. Wolf, "Synthesis of Hard Real-Time Application-Specific Systems," *Design Automation for Embedded Systems*, vol. 4, no. 4, Oct. 1999, pp. 216-242.
6. R.P. Dick and N.K. Jha, "MOGAC: A Multiobjective Genetic Algorithm for Hardware-Software Cosynthesis of Distributed Embedded Systems," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 10, Oct. 1998, pp. 920-935.
7. D. Levine, *Users Guide to the PGAPack Parallel Genetic Algorithm Library*, tech. report, Argonne National Laboratory, Argonne, Ill., 1996.
8. Y. Monnier, J.P. Beauvais, and A.M. Deplanche, "A Genetic Algorithm for Scheduling Tasks in a Real-Time Distributed System," *Proc. 24th Euromicro Conf.,* IEEE CS Press, Los Alamitos, Calif., 1998, pp. 708-714.
9. R.P. Dick, D.L. Rhodes, and W. Wolf, "TGFF: Task Graphs for Free," *Proc. Int'l Workshop Hardware/Software Codesign*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 97-101.
10. A. Chandrakasan, V. Gutnik, and T. Xanthopou-

los, "Data-Driven Signal Processing: An Approach for Energy-Efficient Computing," *Proc. Int'l Symp. Low-Power Electronic Devices*, ACM Press, New York, 1996, pp. 347-352.

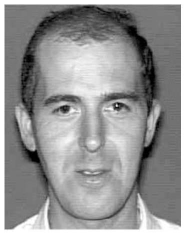
11. A.P. Chandrakasan, S. Sheng, and R.W. Brodersen, "Low-Power CMOS Digital Design," *IEEE J. Solid-State Circuits*, vol. 27, no. 4, Apr. 1992, pp. 473-484.

Peng Yang is a PhD student at IMEC (Interuniversity Microelectronics Center), Belgium. His research interests focus on task-concurrency management for real-time embedded systems. Yang has a BS from Fudan University, China, and an ME from Tsinghua University, China, both in electrical engineering.



Chun Wong is a PhD student at IMEC. His research interests focus on design-time scheduling of concurrent tasks in embedded systems. Wong has a BS in microelectronics from the University of Science and Technology of China and an ME in electrical engineering from Tsinghua University.

Paul Marchal is a PhD student at IMEC. His research interests include dynamic memory management in multiprocessors for multimedia applications. He has an MEng degree in electrical engineering from Katholieke Universiteit Leuven.



Francky Catthoor heads several research groups in high-level and system synthesis techniques and architectural methodologies in the Division of Design Technology for Integrated Information and Telecom Systems at IMEC. He is also a professor of electrical engineering at Katholieke Universiteit Leuven. Catthoor has an MEng degree and a PhD, both in electrical engineering from Katholieke Universiteit Leuven.

Dirk Desmet is a senior researcher at IMEC in the field of reconfigurable platforms. His research interests include object-oriented design methods for embedded systems and reconfigurable systems. Desmet has an MEng in electrical engineering from Katholieke Universiteit Leuven.

Diederik Verkest is a principal scientist in the Technology for Reconfigurable Systems Group at IMEC. His research interests include software-hardware codesign and reconfigurable computing. Verkest has an MEng degree and a PhD in applied sciences, both from Katholieke Universiteit Leuven. He is a member of the Royal Flemish Engineering Society (KVIV) and the IEEE.

Rudy Lauwereins is a professor of electrical engineering at Katholieke Universiteit Leuven and vice president of IMEC. His research interests include computer architectures, implementation of interactive multimedia applications on dynamically reconfigurable platforms, and fault-tolerant computing. Lauwereins has an MEng and a PhD in electrical engineering from Katholieke Universiteit Leuven.

■ Direct questions or comments about this article to Peng Yang, Kapeldreef 75, Leuven, Belgium, B3001, 0032-16-281835; yangp@imec.be.