

A Hierarchical Resource Management Scheme Enabled by the TMO Programming Scheme

K. H. (Kane) Kim

DREAM Lab and EECS Dept.,
University of California, Irvine, USA
khkim@uci.edu

Kee-Wook Rim

Sunmoon University, Korea
rim@sunmoon.ac.kr

Yuqing Li

Microsoft, USA
yuqingli@microsoft.com

Eltefaat Shokri

Affiliated with The Aerospace Corporation, USA
Eltefaat.Shokri@aero.org

Abstract: For cost-effective realization of sizable real-time distributed computing application systems, significant advances in resource allocation are in critical needs. An advanced practical scheme for high-level programming of real-time distributed computing software such as the TMO (Time-triggered Message-triggered Object) scheme introduces the potential for realizing new improved resource allocation approaches. The main reason is because it provides information-rich intuitively appealing timing specifications. However, to properly exploit the potential, a number of issues need to be resolved via extensive research involving both analytical and experimental efforts. A hierarchical resource allocation scheme that becomes enabled by use of the TMO programming model is discussed along with the remaining research issues.

Index Terms: resource allocation, scheduling, real-time, distributed computing, TMO, hierarchical resource management, timing specification, deadline, object, component, TMOSM

1 Introduction

Demands for new real-time (RT) distributed computing (DC) application systems have been on a rapid rise since mid-1990's. In connection with this trend, the demands for better quality-of-services (QoSs) from such systems have also been on the rise. New-generation application systems such as unmanned vehicles and other intelligent robots are some examples of such systems.

Examination of the industry practice and the literature reveals that substantial improvements in QoSs of RT DC systems cannot be realized in absence of tangible advances in some fundamental aspects of RT DC engineering, especially, the *specification and design of action timings*. Research efforts seem particularly needed in two directions.

First, *high-level approaches* for specification and design of RT DC actions need to be explored. The low-level specification approaches which have been practiced over

many years, i.e., those involving explicit manipulation of threads, thread-priorities, and UDP sockets, blur the vision of the software engineer such that he / she cannot see easily the overall picture of the sizable RT DC applications and grasp and analyze the timing requirements that should be imposed on various DC actions. Future RT DC software engineers must be allowed to work with primitive data types which are at least as abstract as *DC objects* that accommodate remote method calls and do not force explicit manipulation of threads and UDP sockets.

Secondly, future large-scale RT DC application systems must be *systematically composed* of rigorously built and / or sufficiently validated building-blocks in order to meet the construction economy and product reliability demands from the societies. Such systematic composition is impractical if the raw materials are of low-level entities such as threads, thread-priorities, and UDP sockets. Action timings must be specified not only in intuitively appealing easily understandable forms but also in forms enabling systematic composition of proper timing behavior of higher-level assemblies.

As an effort motivated by the vision summarized above, the RT DC component model named the *Time-triggered Message-triggered Object* (TMO) was formulated in a skeleton form in early 1990's and has been solidified slowly but steadily since then [Kim97, Kim00, Jen07]. Models of execution engines for TMOs have been developed, some in the form of self-contained operating systems (OSs) layered on DC hardware and others in the form of middleware layered on widely used OS kernels which are in turn layered on DC hardware. The representative middleware-centric execution model is the *TMO Support Middleware* (TMOSM) [Jen07]. In addition, the TMOSL (*TMO Support Library*), consisting of C++ classes which wrap the services of the middleware and collectively serve as an approximation of an idealistic TMO programming language, has been developed. The programming scheme and supporting tools have been used in a broad range of basic research and application prototyping projects in a number of research organizations and also used in an undergraduate course on

RT DC programming at UCI for about four years [<http://dream.eng.uci.edu/eecs123/learn.htm>].

The high-level component-oriented specification and programming approach such as the TMO scheme is aimed for realization of highly dependable RT DC application systems with the labor drastically reduced from that incurred under conventional programming approaches. Application system designs and code become much better "readable". In contrast, understanding sizable RT DC programs composed of threads and UDP sockets is such a tiring endeavor that almost all software engineers are likely to give up on the efforts of analyzing and verifying the "end-to-end" (i.e., from a sensing point in one site to an actuation point in another site) timing behavior, let alone optimizing the overall systems. Therefore, the high-level approaches such as the TMO scheme will go a long way toward instigating fundamental improvements in the reliability of sizable RT DC systems.

A simplistic approach for achieving reliable RT DC systems is to mobilize massive amounts of hardware such that to each simple function, a processor is fully dedicated. In most situations it is not a practical option. For cost-effective realization of sizable RT DC application systems, significant advances in resource allocation are in critical needs. Reasonably high degrees of shared resource utilizations (processors, network connections and bandwidths, peripherals, buffer memory segments, etc) must be facilitated while the timing specifications embedded in RT DC programs are faithfully honored. However, the technical foundation for such resource allocation has not been established in any level near a satisfactory one.

We believe that the biggest obstacle in advancing the technical foundation in a significant way has been the lack of high-level DC programming models. As a result, much of the research has been confined to the cases of dealing with old simple DC models which typically consist of largely, if not completely, independent threads that are often attached fixed priority numbers. From such models, very little knowledge useful for effective resource allocation can be extracted. Therefore, research based on such DC models is not expected to lead to significant advances over whatever occurred in the past. Additional highly restrictive assumptions such as the completion deadline for an RT computational task in each control cycle falling exactly on the beginning of the next control cycle were also often incorporated into the models and as a result, such DC models lost relevance to most of the modern RT DC application situations.

Therefore, research on resource allocation on the basis of advanced practical high-level DC programming models is in order. Good high-level DC models are bound to provide information-rich intuitively appealing timing specifications. This paper deals with the research question of how the timing-related information embedded in the TMO-structured RT DC programs can be exploited

in resource allocation in resource-tight RT DC environments.

Resource allocation in RT DC systems is a complex issue [Wu07]. The resources to be allocated include both local computation resources and communication resources. Sizable RT DC systems are also structured in multiple layers with the DC hardware layer at the bottom, the OS kernel layer next, the middleware layer as the third layer, and then the application layer at the top. Each layer may internally consist of further divided sub-layers. Hierarchical resource allocation is considered a reasonable approach for dealing with such complex situations. In this paper, a hierarchical resource allocation scheme that becomes enabled by use of the TMO programming model is discussed.

2 An Overview of the TMO Scheme

The TMO (*Time-triggered Message-triggered Object*) programming and specification scheme is a general-style RT DC extension of the pervasive object-oriented (OO) design / programming approach [Kim97, Kim00, Kim07]. It has been established to facilitate RT DC software engineering in a form which software engineers experienced in the vast non-RT software field can adapt to with small efforts. Calling the TMO scheme a high-level DC programming scheme is justified by the following characteristics of the scheme:

- (1) No manipulation of processes and threads: Concurrency is specified in an abstract form at the level of object methods. Since processes and threads are transparent to TMO programmers, the priorities assigned to them, if any, are not visible, either.
- (2) Timing requirements need to be specified only in the most natural form of a time-window for every time-triggered method execution, a completion deadline for every client-requested method execution, and a time-window for execution of every significant output action. This high-level expression matches the most closely with the designer's intuitive understanding of the application's timing requirements.

Once the high-level specification of timeliness requirements is registered with the middleware supporting TMOs, then the middleware does its best to meet the specification by using the CPU scheduler and other resource schedulers in the underlying node OS kernel and network infrastructure. Also, the TMO scheme was designed to enable a great reduction of the designer's efforts in guaranteeing timely service capabilities of DC application systems. The TMO incorporates several rules for execution of its components that are intended to make the analysis of the worst-case time behavior of TMOs systematic and relatively easy while not reducing the programming power in any way.

2.1 TMO structure and design paradigms

The key features of the TMO programming scheme are reviewed here. The basic structure is depicted in Figure 1.

(TM1) All time references in a TMO are references to *global time* [Kop97] in that their meaning and correctness are unaffected by the location of the TMO. If GPS receivers are incorporated into the TMO execution engine, then a global time base of microsecond-level precision can be established easily. Within a cluster computer or a LAN based DC system a master-slave scheme, which involves time announcements by the master and exploitation of the knowledge on the message delay between the master and the slave, can be used to establish a global time base of sub-millisecond level precision. A TMO instantiation instruction may contain a parameter which explicitly indicates the required precision of the global time base to be established by the TMO execution engine.

(TM2) TMO is a natural, syntactically minor, and semantically powerful extension of the conventional object(s). TMO is a DC component and thus TMOs distributed over multiple nodes may interact via *remote method calls* and another mechanism (see TM5). Non-blocking types of remote method calls are supported.

(TM3) TMO has been devised to contain only high-level intuitive and yet precise expressions of timing requirements. *Start-time-windows* and *completion deadlines* for object methods are used but no specification in indirect terms (e.g., priority) are required. *Deadlines for result arrivals* can also be specified in the client's calls for service methods.

(TM4) TMO is also an *autonomous active* DC component. Its autonomous action capability stems from one of its unique parts, called the *time-triggered (TT) methods* or the *spontaneous methods (SpMs)*, which are clearly separated from the conventional *service methods (SvMs)*. The SpM executions are triggered upon reaching of the global time at specific values determined at the design time whereas the SvM executions are triggered by service request messages from clients. For example, the triggering times may be specified as "for t = from 10am to 10:50am every 30min start-during (t, t+5min) finish-by t+10min". By using SpMs, *global time based coordination of distributed computing actions (TCODA)* [Kop87, Kop97], can be easily designed and realized.

(TM5) TMOs can use another interaction mode in

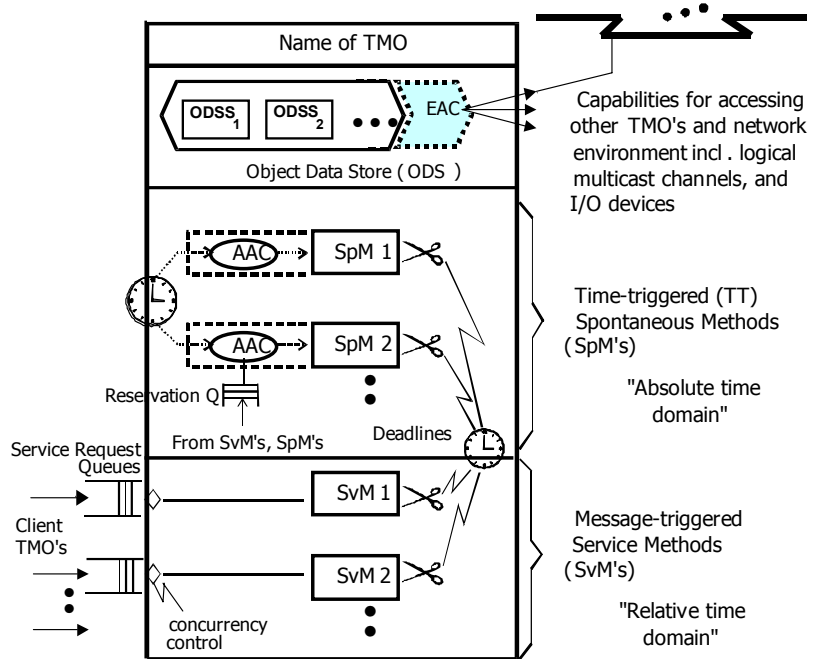


Figure 1. Structure of the TMO (adapted from [Kim97b])

which messages can be exchanged over logical multicast channels of which access gates are explicitly specified as data members of involved TMOs. The channel facility is called the *Real-time Multicast and Memory-replication Channel (RMMC)* [Kim00]. The RMMC scheme facilitates RT publish-subscribe channels in a powerful form. It supports not only conventional *event messages* but also *state messages* based on distributed replicated memory semantics [Kop97].

(TM6) A major execution rule intended to enable reduction of the designer's efforts in guaranteeing timely service capabilities of TMOs is the *basic concurrency constraint (BCC)* that prevents potential conflicts between SpMs and SvMs. The full set of data members in a TMO is called an *object data store (ODS)*. An ODS is declared as a list of *ODS segments (ODSSs)*, each of which is thus a subset of the data members in the ODS and is accessed by concurrently running object-method executions in the *concurrently-reading and exclusive-writing* mode. Basically, *activation of an SvM triggered by a message from an external client is allowed only when potentially conflicting SpM executions are not in place*. Thus an SvM is allowed to execute only if no SpM that accesses the same ODSSs to be accessed by this SvM has an execution time-window that will overlap with the execution time-window of this SvM. The BCC does not reduce the programming power of TMO in any way.

(TM7) An underlying design philosophy of the TMO scheme is that an RT computer system will always take the form of a network of TMOs, which may be produced in a top-down multi-step fashion [Kim97]. Also, TMO

is capable of representing all conceivable practical RT and non-RT applications.

2.2 TMO Support Middleware (TMOSM), the core part of the TMO execution engine model

TMO programming has been enabled without creation of any new language or compiler. Instead, a middleware model called the TMOSM (*TMO Support Middleware*) provides execution support mechanisms and can be easily adapted to a variety of commercial hardware + kernel platforms, e.g., PCs or similar hardware with Windows XP, Windows CE, or Linux [http://dream.eng.uci.edu/TMOdownload/, Jenk07]. TMOSM uses well-established services of widely used OS kernels, e.g., process and thread support services, short-term scheduling services, and low-level communication protocols, in a manner transparent to the application programmer.

While devising the TMOSM architecture, an emphasis was on making both the analysis of the worst-case time behavior of the middleware and the analysis of the execution time behavior of application TMOs as easy as possible without incurring any significant performance drawback. As a result, use of mechanisms such as semaphore which leads to frequent blockings of threads inside the middleware was avoided almost completely and instead, the *Non-Blocking Writer* mechanism [Kop97] and the *Non-Blocking Buffer* (NBB) mechanism [Kim06], were used extensively.

Our experiences indicate that even this middleware extension of a general-purpose OS kernel such as Windows XP can support application actions with the 10ms-level timing accuracy. A Windows CE based prototype and a Linux based prototype of TMOSM are under continuous optimization with the goal of supporting application actions with better-than-10ms-level timing accuracy.

As depicted in Figure 2, within TMOSM, the innermost core is a super-micro thread called the WTST (*Watchdog Timer & Scheduler Thread*). It is a "super-thread" in that it runs at the highest possible priority level. It is also a "micro-thread" in that it manages the scheduling / activation of all other threads in TMOSM. Even those threads created by the node OS kernel before TMOSM starts are executed only if WTST allocates some time-slices to them. Therefore, WTST is in control of

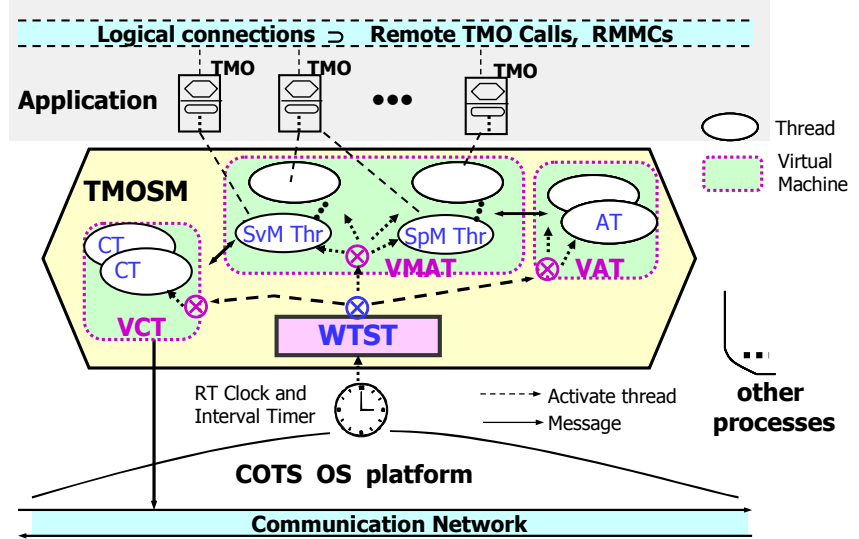


Figure 2. Virtual machines and threads in TMOSM

the processor and memory resources with the cooperation of the node OS kernel.

WTST leases processor and memory resources to three *virtual machines* (VMs) in a time-sliced and periodic manner. Each VM can be viewed conceptually as being periodically activated to run for a time-slice. Each VM is responsible for a major part of the functions of TMOSM. Each VM maintains a number of *application threads*. In fact, whenever WTST assigns a time-slice to a VM, the VM in turn passes the time-slice onto one of the application threads belonging to itself. The component in each VM that handles this "time-slice relay" is the *application thread scheduler*.

The set of VMs is *fixed at the TMOSM start time*. One iteration of the execution of a specified set of VMs is called a *TMOSM cycle*. For example, one TMOSM cycle may be: VCT VMAT VAT VMAT. Here threads in VMAT are assigned to various TMO-method executions whereas VCT has the monopoly in managing network message communications and provides messenger services to VMAT and any other VM. Threads in VAT are used to run auxiliary application functions and unused portions of the time-slices given to VAT are donated to those native threads created by the node OS kernel. This architecture led to relatively easily analyzable timing behavior of TMOSM.

We judge that this VM based architecture of TMOSM enables effective use of multi-core microprocessors. Each VM is considered a perfect candidate to which a core can be dedicated. Moreover, with multi-core microprocessors, the number of VMATs can be increased without slowing down the execution of each TMO.

3 Hierarchical Resource Management in TMOSM

In the TMO-structured RT DC environments, one can envision resource allocations occurring at three different levels.

- (1) At the networked system level: The *TMO Network Configuration Manager* (TNCM) instantiation in each node cooperates with its peers, i.e., TNCM instantiations in other nodes, for distributing TMOs to proper nodes and resolving the competition among the nodes for shared execution resources such as network bandwidth.
- (2) At the node level: WTST allocates proper processor cycles (to be precise, time-slices) to each virtual machine such as VMAT, VCT or VAT. VCT is also instructed on how to use the acquired network bandwidth to support VMAT and VAT.
- (3) At the virtual machine (VM) level: The intra-VM scheduler allocates resources to each ready thread in its domain.

3.1 At the networked system level: Assignment of application workloads and network access windows to DC nodes

The resource allocation at this level involves two kinds of actions: (1) efficient deployment of TMOs in computing nodes and (2) the allocation of shared network bandwidth among computing nodes.

The message communication delays within an RT DC system must be bounded since otherwise, timely responses of the application system cannot be guaranteed. A proven way to achieve bounded communication delays in an isolated Ethernet environment is to make DC nodes to access the network in a TDMA fashion as illustrated in Figure 3. The TMOSM cohort in a computing node thus sends messages to the communication network during its communication time-slots only. Multiple communication slots in one TDMA cycle may be allocated for a particular node as shown in Figure 3 and Node 1 there owns $2/5 = 40\%$ of the overall bandwidth. TMOSM has been designed to find a specification of TDMA slot allocations from the file with the name "config.ini".

Moreover, the network bandwidth sharing among the

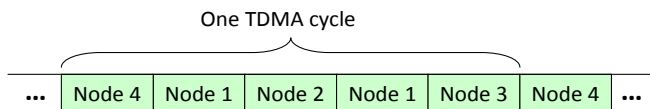


Figure 3. TDMA cycles maintained by TMOSM cohorts

nodes can be adjusted by the TNCM based on run-time information. There are two cases:

- (1) If a computing node crashes, the communication time-slot assigned to the failed node can be reassigned to another healthy node. To achieve this, timely fault detection of each computing node is necessary.
- (2) The communication slot of a node can also be donated to another node if there is no future message transmission from the former node expected. For example, if there is no registered SvM in the node and the time-windows for all the SpMs have expired, the communication slot assigned to the node can be reassigned to another node.

At present we take the conservative approach of identifying at design time all possible TMO distributions to be made at run time and preventing any TMO distribution which cannot be assured to be sufficiently safe.

3.2 At the node level: Assignment of processor time-slices to VMs and network access windows to different VCT services

The resource allocation at this level involves two kinds of actions: (1) the allocation by WTST of processor time-slices to each individual VM and (2) the allocation by VCT of the acquired network access windows to different services.

TMOSM adopts a two-level scheduling scheme: (L1) VM scheduling and (L2) application thread scheduling. The initial schedule of VMs in a node is defined at the design time in a configuration file "config.ini". Each time a new time-slice begins, WTST selects the next VM -- one of VCT, VAT, and VMAT. In the case where the VM execution cycle in node 1 is specified as VMAT VAT VMAT VCT, 2/4 of the time-slices are given to VMAT. Then, the *application thread scheduler* belonging to the VM is called in to select the next application thread to be executed.

VCT is the sole VM in TMOSM which directly accesses the network. Other VMs may order message transmission services to VCT via relevant message queues. There may be (m1) messages exchanged among the VMAT peers in DC nodes and (m2) those among the VAT peers in addition to (m3) the messages originated from a VCT cohort and ending at a VCT cohort. The "config.ini" file may thus specify the *transmission bandwidth usage* (TBU) for each class of messages. In a sense, it indicates how many percents of the bandwidth owned by VCT are reserved for supporting the target VM. One can also view it as subdividing a TDMA slot further into sub-slots. For example, consider that the VCT's *transmission service cycle* in node A is VUSER1

VUSER2 VMAT VAT VCT, where VUSER1 and VUSER2 are two customized VMs, and that the TBU specification is 0 10 50 0 40. It indicates that 50% of the TDMA slots owned by the VCT should be used for serving VMAT, 10% for VUSER2, and 40% for its own transmission to its peers.

At run-time, the processor time-slices allocated to a VM, or their unused portions, can be donated to other VMs if there is no ready thread in the former VM or the former VM does not have any message to send out.

3.3 At the VM level

Resource allocation at this level involves primarily the scheduling of application threads (supporting the admitted TMOs) based on the deadlines or laxities associated with the program-segments being carried by the threads.

In addition, it can in principle perform the secondary function of supporting the resource manager at the networked system level on deciding whether to admit a TMO or not. It can check if the admission is safe from the viewpoint of the VM. When the VM recommends admission of a TMO, the availability of the processor cycles and network access windows needed to support the TMO is guaranteed. However, we feel that the currently existing technical foundation is too weak to enable such fully automated admission decision. Therefore, identification of various possible safe distributions of TMOs at design time through extensive testing aided by some analyses is the current practical recourse suggested to TMO designers.

Nevertheless, for future research on the prospects of enabling dynamic admission of TMOs into the TMO distributions not anticipated before, we consider it worth pursuing the direction briefly sketched below. The processor cycle requirements that a TMO imposes on VMAT can be estimated on the basis of the timing specifications embedded in TMOs plus the *tight isolated execution time bound* (TIETB) of each method in the TMO. Here a TIETB of a TMO-method is an execution time bound for the cases where the subject method runs alone in the host node without any other TMO or TMO-method co-resident in the same host node in an active mode.

The timing specifications in the current TMO model include (1) autonomous activation conditions (AACs) associated with SpMs, which include the *guaranteed completion time* (GCT, also called the *guaranteed execution time bound*), (2) GCTs of SvMs, and (3) the *maximum invocation rate* (MIR) associated with each SvM. Currently, TMO designers are required to specify

a GCT for each SpM and one for each SvM but not required to provide a TIETB for each TMO-method. TIETBs are expected to be provided by a tool(s) which analyzes the program code as well as performing test-runs. Such a tool is under development on the basis of a promising hybrid approach [Im06]. A TIETB can be obtained from a hybrid of exhaustive measurements of isolated execution times and program code analyses. A GCT of a TMO-method should of course be set to be greater than the TIETB of the method. By setting GCTs of all TMO-methods, the efforts needed for guaranteeing the timeliness of the top-level TMO-methods which interact directly with the application environments, become greatly reduced. In setting GCTs in a TMO, the TMO designer reflects the possibility of the subject TMO being co-resident with some other TMOs in the same node. MIR is the maximum rate at which the server TMO containing the SvM can receive service requests from client TMOs, e.g., 10 invocations per 100 milliseconds.

A simple approach for estimating a tight upper bound on the processor cycle requirements is to compute the bound on the CPU utilization of the target TMO by use of the following formula.

$$\sum_{k=0}^{\#ofSpM} \frac{TIETB(SpM_k)}{Period(SpM_k)} + \sum_{k=0}^{\#ofSvM} [TIETB(SvM_k) * MIR(SvM_k)]$$

Here MIR is a normalized value expressed in terms of the number of invocations allowed per unit time. The first part of the above formula defines an upper bound on the CPU utilization by all SpMs in the target TMO, and the second part defines an upper bound on the CPU utilization by all SvMs in the target TMO.

Therefore, if the current TMOSM is extended to accommodate a TIETB of each TMO-method provided as a part of the method registration parameters, then it can tell at the time of the TMO registration (which follows the registrations of all the member methods) whether the TMO can be safely admitted or not. However, there are some inevitable rooms for errors (e.g., false alarms) in this decision and thus the TMO designer should be provided a means of enforcing admission. Much further research is needed in this area.

TMO provides action timing requirements with which a variety of deadline-driven thread scheduling polices can be easily implemented. However, simple earliest-deadline-first (EDF) scheduling based on GCTs of TMO-methods leads to one undesirable consequence: when TMO-methods that can all go in parallel produce *intermediate outputs*, i.e., outputs produced in the middle of their executions in contrast to those produced at the end, intermediate outputs of some TMO-methods are greatly

delayed while intermediate outputs of some others are produced much earlier. Major examples of intermediate outputs include sending service requests to another TMO, generating commands to the actuator devices, and updating shared variables. A practical approach to avoid such phenomenon is to attach completion deadlines to the intermediate output actions and let TMOSM reflect those deadlines in scheduling application threads. To be practical such deadlines should be mechanically derived from the GCTs for TMO-methods supplied by the TMO designer.

A simple approach for determining a deadline to be attached to an intermediate output is as follows. Assume that a TIETB of the method-segment that starts from the beginning of the method and ends at the finishing point of the intermediate output is D1 and a TIETB of the method-segment starting after the intermediate output and covering the rest of the method is D2. Then the deadline for the intermediate output can be set to:

$$\text{GCT} \times D1 / (D1 + D2).$$

Such an approach for setting intermediate deadlines was discussed previously, e.g., [Zha05]. The TMO scheme makes its practical realization feasible. A reasonably effective method and tool for determining a TIETB of a method-segment that does not involve any OS (kernel + middleware) call have been developed [Im06]. Their extensions to cover the method-segments that involve some OS calls are expected to be completed later this year. Once systematic derivation of TIETBs of method-segments becomes possible, the application thread scheduling that was discussed above and is considered to be significantly more efficient than what is in use in industry today, or its enhancement, will become a firm fixture in the TMO execution facility.

4 Conclusion

The framework for hierarchical resource management that is enabled by the TMO programming scheme has been discussed. Although it offers potentials for much more cost-effective resource allocation than what is widely practiced in industry today, its practical realization requires developments of methods and tools for deriving TIETBs of TMO-methods and method-segments. In addition, much further research is needed to identify cost-

effective approaches for scheduling threads carrying method-segments.

Acknowledgment: The work reported here was supported in part by the NSF under Grant Numbers 03-26606 (ITR) and 05-24050 (CNS).

References

- [Imc06] Im, C.S., and Kim, K.H., "A Hybrid Approach in TADE for Derivation of Execution Time Bounds of Program-Segments in Distributed Real-Time Embedded Computing", *Proc. ISORC 2006*, Gyeongju, Korea, April, 2006, pp.408-418.
- [Jen07] Jenks, S.F., Kim, K.H., et al., "A Middleware Model Supporting Time-Triggered Message-Triggered Objects for Standard Linux Systems", *Real-Time Systems - The International Journal of Time-Critical Computing Systems*, Vol. 36, April 2007, pp.75-99.
- [Kim97] Kim, K.H., "Object Structures for Real-Time Systems and Simulators", *IEEE Computer*, Vol. 30, No.8, August 1997, pp. 62-70.
- [Kim00] Kim, K.H., "APIs for Real-Time Distributed Object Programming", *IEEE Computer*, Jun. 2000, pp.72-80.
- [Kim06] Kim, K.H., Colmenares, J., and Rim, K.W., "Efficient Adaptations of the Non-blocking Buffer for Event Message Communication between Real-Time Threads", *Proc. ISORC 2007*, Santorini, Greece, May, 2007, pp.29-40.
- [Kim07] Kim, K.H., and Colmenares, J., "Maximizing Concurrency and Analyzable Timing Behavior in Component-Oriented Real-Time Distributed Computing Application Systems", *KIISE Journal of Computing Science and Engineering (JCSE)*, Vol.1, No.1, Sep. 2007, pp. 56-73 (downloadable from <http://jcse.kiise.org>).
- [Kop97] Kopetz, H., *Real-Time Systems: Design Principles for Distributed Embedded Application*, Kluwer Acad. Pub., ISBN: 0-7923-9894-7, Boston, 1997.
- [Wu07] Wu, H., Balli, U., Ravindran, B., Anderson, J., and Jensen, E. D., "Utility Accrual Real-Time Scheduling Under Variable Cost Functions", *IEEE Trans. on Computers*, Vol.56, No.3, Mar. 2007, pp. 385-401.
- [Zha05] Zhang, J.Y., DiPippo, L., Fay-Wolfe, V., Bryan, K., and Murphy, M., "A Real-Time Distributed Scheduling Service for Middleware Systems", *Proc. WORDS 2005*, Sedona, AZ, Feb. 2005, pp.59-65.