Non-preemptive Scheduling of Real-Time Threads on Multi-level-context Architectures

Jan Jonsson[1], Henrik Lönn[1], and Kang G. Shin[2]

[1] Dept. of Computer Engineering, Chalmers, S–412 96 Göteborg, Sweden
{janjo,hlonn}@ce.chalmers.se
[2] Dept. of Elec. Engr. and Computer Science, Univ. of Michigan, Ann Arbor
kgshin@eecs.umich.edu

Abstract. This paper addresses the problem of how to schedule periodic, real-time threads on a class of architectures referred to as *multi-level-context* (*MLC*) architectures. Examples of such architectures are real-time operating systems with support for user- or kernel-level threads, and multithreaded microprocessors endowed with on-chip contexts. A common feature of these architectures is that they provide support for the administration of threads within contexts at different levels of abstraction. Therefore, the cost for switching between threads will depend on the affinity of their corresponding contexts. The main contributions of this paper are to demonstrate (i) how the scheduling performance for off-line scheduling on MLC architectures can benefit from an integrated heuristic that is cognizant of both the time-criticality of a thread and the current context affinity; and (ii) how the predicted performance for on-line scheduling on MLC architectures can benefit from an off-line schedulability test that accounts for variations in the context affinity.

1 Introduction

Assessing the impact of architectural properties on the schedulability of a real-time application has become a very important issue in the real-time research community. Most modern microprocessor architectures are endowed with such mechanisms as caches and instruction pipelines whose temporal behavior are very difficult to predict *a priori*. Despite many recent successful attempts to model the behavior of these mechanisms [1, 2], real-time scheduling research has not been able to keep pace with the progress in microprocessor design. This has led to an unfortunate gap between real-time scheduling theory and its practice.

An important remaining problem is to analyze the effects of *context switch* operations on the schedulability analysis. While traditional real-time schedulability tests [3] simply assumed that a context switch can be made at a negligible cost, more recent tests [4] are capable of incorporating a non-negligible context-switch cost. Unfortunately, these schedulability tests are based on the assumption that the cost for a context switch is constant and independent of the underlying thread-invocation pattern. This assumption does not hold for an important emerging class of architectures referred to as *multi-level-context*

(*MLC*) architectures. A common feature of these architectures is that they provide software and/or hardware support for the execution of multiple *threads* in the application. The administration of threads requires support for *contexts* at different levels of abstraction, that is, with different amounts of execution states to maintain. When there are more than one context level, the cost for switching between two threads will depend on the affinity of their contexts. Since the context-switch scenario cannot typically be predicted in advance, existing techniques for schedulability analysis can only be used with an MLC architecture at the expense of overly-pessimistic assumptions on the context-switch cost. This will dramatically reduce the usefulness of an off-line schedulability test.

Many existing real-time systems can be classified as MLC architectures, *e.g.*, real-time operating systems with support for user- or kernel-level threads [5]. Also, thread-level parallelism is believed to be the only alternative to maintain the performance growth of microprocessors [6]. Because one can predict an increasing demand for computing power in modern real-time applications [7], it is very likely that multithreaded architectures will also be used widely in future real-time systems. In order to analyze the real-time performance of such architectures, their degree of run-time predictability must be thoroughly assessed.

In this paper, we address the problem of non-preemptive scheduling of periodic, real-time threads on a uniprocessor MLC architecture. Non-preemptive scheduling is the natural choice for many systems since (i) uncontrolled preemption can cause extra context switches which may jeopardize the schedulability, (ii) exclusive access to shared resources is guaranteed which eliminates much of the need for synchronization and its associated overhead, and (iii) worst-case execution time analysis becomes easier since the instruction stream of a thread can execute without interference by other threads. Using a set of randomly-generated application thread sets and various context-configuration scenarios, we study the performance of a set of heuristic algorithms using the *success ratio* as the performance measure — that is, the ratio of the number of successfully-scheduled thread sets to the total number of evaluated thread sets.

Despite the growing interest in the MLC architectures, very few results on thread scheduling on them have been reported. The main intent of this paper is, therefore, to make the following two contributions:

C1. For real-time systems employing a time-driven thread dispatching policy, we demonstrate an integrated polynomial-time scheduling heuristic — cognizant of both time-criticality and the current context affinity — that significantly outperforms other heuristics when used with MLC architectures.

C2. For systems employing a priority-driven thread dispatching policy, we propose a sufficient schedulability test for the on-line EDF algorithm that makes far better predictions than existing tests when used with MLC architectures.

We start in Sect. 2 to elaborate on the problem of scheduling on MLC architectures, and then continue in Sect. 3 to introduce the assumed system models and the terminology used. In Sect. 4, we present an experimental evaluation of our proposed techniques. In Sect. 5, we discuss related work and give a brief description of future work. Finally, we summarize our findings in Sect. 6.

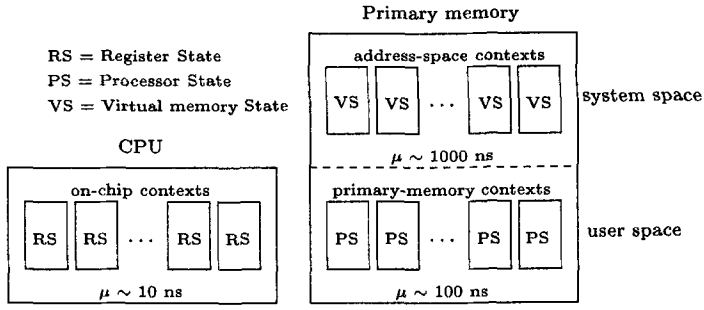

Fig. 1. Multi-Level-Context architecture.

2 Background and problem statement

In an MLC architecture, the schedulable entities are *threads*, streams of control that represent stretches of executable code in an application. Each thread executes within multiple levels of *contexts*. Each context is a locus of control that describes the current execution state of the thread at a certain abstraction level. When switching between two threads, a *context switch* must take place so as to exchange the active execution state. The cost for a context switch depends on the *context affinity* of the threads, which is defined as the amount of execution information shared between the threads.

Depending on the level of context abstraction, the cost for a context switch consists of different components. An *on-chip* context is typically found in multiple-context, multithreaded architectures where the registers are either duplicated in hardware [8, 9], or partitioned in software [10], to hold multiple active threads. The cost for switching between two on-chip contexts on a pipelined processor is in the range of 5–10 clock cycles. The *primary-memory* context contains processor user registers (*e.g.*, in a thread control block). The total cost for these operations is typically in the range of 100–200 clock cycles for a RISC processor with a large number of user registers. An *address-space* context is typically found at the operating system level. The cost for switching between address-space contexts includes the cost for updating virtual-memory mechanisms (*e.g.*, TLB and MMU), operations that can take thousands of clock cycles to perform.

Fig. 1 illustrates how the context-switch cost μ relates to the different context levels in a typical MLC architecture. Apparently, depending on how the application threads are organized with respect to the contexts levels, the cost for a context switch can vary by as much as an order of a magnitude at run-time. Thus, depending on the way the application has been partitioned into threads, and the invocation pattern of the threads at run-time, the cost for switching between two threads will vary with the current context affinity of the threads, which, in turn, is a function of time. Unless such variations are taken into account during off-line analysis, an expensive context switch at run-time may cause a thread, and hence the application, to miss its deadline. The objective of this paper is to find effective real-time scheduling strategies that are robust with respect to such variations in the context-switch cost.

3 System models and terminology

3.1 Processor model

We assume a processor model similar to the MIT SPARCLE processor [8] where the register windows are utilized in pairs to form independent, non-overlapping on-chip contexts. We denote the switch costs for the on-chip, primary-memory, and address-space context levels by μ_ℓ, μ_g, and μ_a, respectively.

3.2 Thread model

We consider a real-time application that consists of a set $\mathcal{T} = \{\tau_i : 1 \leq i \leq n\}$ of threads. The threads are assumed to be part of tasks that represent major computations in the application. Each thread $\tau_i \in \mathcal{T}$ is characterized by a 5-tuple $\langle c_i, \phi_i, d_i, p_i, \vartheta_i \rangle$. The *worst-case execution time* c_i is an upper bound of the execution time of the thread and includes various architectural overheads such as the cost for cache misses and pipeline hazards, as well as the on-chip context-switch cost μ_ℓ. The *phasing* ϕ_i is the earliest time by which the first invocation of the thread will occur. The *relative deadline* d_i is the amount of time within which the thread must complete its execution. The *thread period* p_i is the time interval between two consecutive invocations of the thread. The *context* ϑ_i is the primary-memory state within which the thread will execute.

We organize the threads in *period classes*, with one class π_k for each distinct period used by threads in $\mathcal{T}$. We denote the collection of period classes by Π, the corresponding period of π_k by $p(\pi_k)$, and the $n_t(\pi_k)$ threads in π_k by $\tau_{\pi_k,1}, \tau_{\pi_k,2}, \ldots, \tau_{\pi_k,n_t(\pi_k)}$. It is also useful to collect the threads sharing the same primary-memory context in *context groups*, with one group $\theta_{\pi_k,r}$ for each distinct context used by threads in π_k. We denote the collection of $n_g(\pi_k)$ context groups in π_k by $\Theta_{\pi_k} = \{\theta_{\pi_k,1}, \theta_{\pi_k,2}, \ldots, \theta_{\pi_k,n_g(\pi_k)}\}$, the corresponding context of $\theta_{\pi_k,r}$ by $\vartheta(\theta_{\pi_k,r})$, and the number of threads in $\theta_{\pi_k,r}$ by $n_t(\theta_{\pi_k,r})$.

Precedence constraints between threads in $\mathcal{T}$ are represented as follows. If thread τ_j cannot begin its execution until thread τ_i has completed its execution, we write $\tau_i \prec \tau_j$. In this case τ_i is said to be a *predecessor* of τ_j, and, conversely, τ_j a *successor* of τ_i. A thread which has no predecessors is called an *input thread*, and a thread which has no successors is called an *output thread*. A *thread chain* is defined as an input thread followed by a series of parallel/serial threads and terminated by one output thread.

3.3 Execution model

The non-preemptive execution of application threads is administrated by a run-time dispatcher. When a thread is selected by the dispatcher, it begins its execution and runs without preemption until it voluntarily transfers control of the processor back to the dispatcher. To understand the behavior of the dispatcher, we start by describing the dynamic behavior of a thread τ_i. Let τ_i^k denote the k^{th} invocation of τ_i, $k \in \mathbf{Z}^+$. The *absolute arrival time* $a_i^k = \phi_i + p_i(k-1)$ is the

earliest time by which τ_i^k is allowed to start its execution. The *absolute deadline* $D_i^k = a_i^k + d_i$ is the latest time by which τ_i^k must complete its execution.

Now, assume that the dispatcher is in the process of selecting a new thread to schedule at time t. We then make the following definitions.

Definition: A thread τ_i is *ready* at time t if its immediate predecessors have finished their execution.

Definition: A thread τ_i is *released* at time t if it is ready, $a_i \leq t$, and the thread has received all necessary data from its immediate predecessors.

Definition: A thread is called a *continuation thread* if it is ready and belongs to the same context group as the most-recently-executed thread.

Definition: A thread is called a *completion thread* of period class π_k if it is executed last among all threads in π_k within one particular invocation.

Definition: A thread is said to be *returning* from period class π_j to period class π_k if it belongs to π_j, is the most-recently-executed thread, and the next thread that executes belongs to π_k.

Definition: A scheduling policy that allows itself to insert idle time slots instead of executing a ready thread is said to use *inserted idle time.*

4 Experimental evaluation

4.1 Quality assessment

Our primary performance measure is the *success ratio*, defined as follows. If a real-time scheduling strategy can aid in finding feasible schedules for x of the y thread sets considered, its success ratio is (x/y). We choose to study the delivered performance as a function of the **context-switch cost** (μ_g) and the **context affinity** (η). The former represents the cost for performing a switch to another primary-memory context and reflects two important characteristics of the system: (i) the number of on-chip registers that must be exchanged, and (ii) the granularity (in terms of execution time) of the application threads. The latter represents the probability that two arbitrarily-chosen threads in the generated thread set share the same primary-memory context. This parameter allows us to observe the effect of the number of primary-memory contexts in the system on scheduling performance. The affinity ranges from $\eta = 0.0$ (threads execute in separate contexts) to $\eta = 1.0$ (threads execute in a common context).

4.2 Architecture

Similar to the SPARCLE processor, we assume that there are four on-chip contexts. We also assume that there are eight primary-memory contexts. When a primary-memory context switch occurs, we assume that all on-chip contexts are reloaded and that all memory references hit in the cache during the context switch. The primary-memory context-switch cost ranges from 0% (no overhead) to 25% (significant overhead) of the mean thread execution time, c_{mean}. We also assume that all threads execute within one single address-space context. Any overhead associated with interrupt handling and thread dispatching in the run-time system is assumed to be negligible in our experiments.

4.3 Workload

To evaluate[1] the robustness of each heuristic in the presence of varying context-switch costs, we attempted to schedule 1024 randomly-generated sets of threads, each containing 40–50 threads organized in four chains of threads. Each chain of threads was then randomly mapped to one of the eight primary-memory contexts, as determined by the context affinity, η. The default value of η is 0.25. The chains were generated in such a way that the parallelism of the threads sharing one specific context could be fully exploited on the on-chip contexts. Individual thread execution times were chosen at random assuming a uniform distribution in the range of 15 to 25 μs, and a mean execution time $c_{mean} = 20$ μs.

All threads in a chain were assigned a common period, chosen at random (with uniform probabilities) from four period classes with the corresponding periods $p(\pi_1) = 0.8p$, $p(\pi_2) = 1.0p$, $p(\pi_3) = 1.2p$, and $p(\pi_4) = 1.6p$. The base period p was chosen so that the total utilization U of the processor was kept at approximately 85%. We assume for each thread τ_i that $\phi_i = 0$ and $d_i = p_i$.

4.4 Pre-run-time scheduling for a time-driven system

We first investigate the time-driven dispatching policy where run-time thread dispatching is performed using a table with explicit start and finish times of each thread. The table of thread invocations (the schedule) is constructed off-line. The off-line scheduling heuristics under investigation in this experiment are all variations of the *list scheduling* [12] strategy, which uses inserted idle time.

Fig. 2a plots the performance of the evaluated heuristics as a function of the primary-memory context-switch cost μ_g. The figure presents the results for five different heuristics. We first discuss the performance of two simple list-scheduling strategies: the earliest-deadline-first (EDF) and earliest-start-time (EST) heuristics. In these strategies, the next thread to schedule in the priority list is the one with the closest (absolute) deadline and the one that can begin execution soonest (accounting for any precedence or periodicity constraints), respectively. The plots indicate that the EDF heuristic does not perform well under the list-scheduling policy. In fact, EDF is not capable of scheduling more than 30% of all generated thread sets, regardless of context-switch cost and context affinity. Moreover, the success ratio drops as the context-switch cost increases owing to the fact that the deadline of a thread does not contain any information as to whether a context switch will be taken or not should the thread be scheduled next. In contrast, the plots for the EST heuristic indicate that EST performs better when the context-switch cost is significant, than when it is zero. In fact, the success ratio increases from 30% to approximately 50% when the context-switch cost increases from zero to 5%. The explanation for this behavior is as follows. As soon as the context-switch cost is non-negligible, the earliest start time of a thread indicates whether a context switch will be taken or not should the thread be scheduled. Hence, the list-scheduling algorithm can take advantage of this information when selecting the next thread to schedule.

[1] All modeling and simulations were performed within the GAST [11] framework.

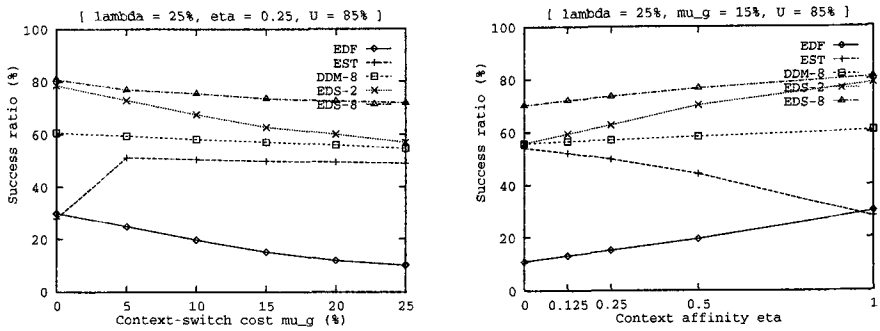

Fig. 2. Success ratio for time-driven dispatching as a function of a) the primary-memory context-switch cost μ_g assuming a context affinity $\eta = 0.25$; b) the context affinity η assuming a primary-memory context-switch cost $\mu_g = 15\%$.

In the dynamic deadline modification (DDM) technique [13], non-continuation threads have their original deadlines temporarily modified during thread selection so as to favor continuation threads. The temporary deadline of a non-continuation thread τ_i is $\hat{D}_i = D_i + k$, where D_i is the original deadline of the thread and k is an offset factor. In our experiments, we have used the DDM with an offset factor $k = 8$ (denoted in the figure by DDM-8). Fig. 2a clearly indicates that this approach is successful in increasing the success ratio.

An interesting choice of heuristic would be to integrate the time-criticality-cognizant traits of the EDF heuristic with the context-switch-cognizant traits of the EST heuristic. To this end, we define the *earliest-deadline-and-start-time* (EDS) heuristic, where the priority function for a thread τ_i is the weighted sum $D_i + w \times t_{est}$, where D_i is the (absolute) deadline of the thread, t_{est} is the earliest-start-time of the thread, and w is a weight factor. Fig. 2a shows the plots for the EDS heuristic with a weight factor $w = 2$ and 8 (denoted by EDS-2, and EDS-8, respectively). As indicated by these plots, the integrated approach gives a significant boost in performance. Initially, the success ratio increases to more than 75% for a negligible context-switch cost, and, as the context-switch cost increases, the plots show the potential of the integrated approach — in particular when the weight factor increases. For example, a weight factor $w = 8$ gives a very robust behavior as compared with the original EDF and EST heuristics[2].

Fig. 2b plots the performance of the evaluated heuristics as a function of the context affinity η. The plots indicate that EDS-8 is very robust with respect to variations in the context affinity. Also, note the anomalous behavior of EST as the context affinity increases. While the other heuristics benefit from an increase in the context affinity because the degree of sharing increases, EST performs worse as the earliest start time of a thread provides less and less information on whether a primary-memory context switch will be taken or not.

[2] Note that an increase in the weight factor not automatically implies an increase in performance. Too high a weight factor would give the "EST" part of the heuristic too strong an influence, which would cause the performance to reduce to that of original EST. We found that $w = 8$ through $w = 32$ gives the overall best performance.

4.5 Run-time scheduling for a priority-driven system

In this section, we explore a priority-driven dispatching scheme. Threads are kept in a list at run-time, ordered by their priorities. Whenever the dispatcher selects a thread to execute, it selects the one that has the highest priority in the list. Inserted idle time is not used for this dispatching scheme, which means that only released threads will be considered during thread selection. We assign thread priorities according to the earliest-deadline-first (EDF) policy.

Now, since we are using an on-line scheduling algorithm, it is necessary to verify, off-line, that all thread deadlines will be met at run-time with the employed dispatching policy. To this end, we use the schedulability test proposed by Zheng and Shin [14], modified to account for context-switch costs.

Theorem 1 (Zheng and Shin). *In the presence of non-real-time threads, a set of real-time threads* $\mathcal{T} = \{\tau_i : 1 \leq i \leq n\}$ *with* $d_i = p_i$ *are schedulable on a uniprocessor with a primary-memory context-switch cost* μ_g *under the non-preemptive EDF policy if the following conditions hold:*

$$1.\ \sum_{\tau_i \in \mathcal{T}} (c_i + \mu_g)/p_i \leq 1.$$

$$2.\ \forall t \in S,\ \sum_{\tau_i \in \mathcal{T}} \lceil (t - p_i)/p_i \rceil^+ (c_i + \mu_g) + c_p \leq t,$$

$$\text{where } S = \cup_{\tau_i \in \mathcal{T}} \{t : t \leq t_{max},\ t = p_i + mp_i,\ m \in \mathcal{N}\},$$

$$t_{max} = \max\{p_1, \ldots, p_n\}$$

One obvious problem with the schedulability test in Theorem 1 is that it assumes that one primary-memory context-switch is taken for each thread invocation. This is not a reasonable assumption since, on an MLC architecture, multiple threads execute within shared contexts. Thus, many primary-memory context switches accounted for in Theorem 1 will, in fact, never be taken at run-time. Unfortunately, with the current run-time policy, it is difficult to calculate tight upper bounds on the number of context switches unless we completely predict the thread invocation pattern in advance. In order to make better predictions on the context-switch scenario, we introduce a degree of determinism by replacing the original EDF policy with a similar scheduling policy that we call the EDF* policy. The new scheduler behaves exactly like the original EDF scheduler except when choosing among released threads with the same deadline. Recall that the EDF rule allows for an arbitrary choice among threads with the same deadline. We will exploit this feature to enforce the precedence constraints on the threads. When there are multiple released threads with the nearest deadline, the EDF* policy will choose a continuation thread whenever one exists. This decision will be enforced at run-time by complementing the dynamic-priority policy with an additional static thread priority that will guarantee that continuation threads will be favored before other threads. The actual static-priority assignment is performed according to the CLASSIFY algorithm listed in Fig. 3. The main purpose of this algorithm is to guarantee that the execution of threads in the same period class gives rise to as few context switches as possible.

Algorithm CLASSIFY:

1. create a set Π of period classes: one class π_k for each distinct thread period;
2. create one local context group $\theta_{\pi_k,r}$ for each distinct thread context in π_k;
3. enumerate context groups within each period class according to significance;
4. initialize a variable z with the highest possible priority;
5. **for** { each $\pi_k \in \Pi$, shortest period first } **loop**
6. **for** { each $\theta_{\pi_k,r} \in \pi_k$, most significant first } **loop**
7. assign to all threads in $\theta_{\pi_k,r}$ a priority equal to z;
8. decrease the priority variable z;
9. **end loop**;
10. **end loop**;

Fig. 3. The period-based classification algorithm.

Now, to find a schedulability test that is less pessimistic than the one in Theorem 1, we analyze the period classes one by one and determine, for each period class π_k, the number of context switches caused by (i) threads being *invoked* in π_k, and (ii) threads *returning* from another period class to π_k. To this end, we use Lemma 1. The proof can be found in detail in [15].

Lemma 1. *Assuming that the non-preemptive EDF* policy and the* CLASSIFY *algorithm are used, an upper bound $n_c(\pi_k)$ on the total number of primary-memory context switches experienced while executing the threads in π_k during the time interval $p(\pi_k)$ can be found as follows.*

$$n_c(\pi_k) = \min\{n_t(\pi_k),\ n_g(\pi_k) + \sum_{\pi_j \in \Pi\ :\ p(\pi_j) < p(\pi_k)} \min\{\hat{n}_t(\pi_k, \pi_j),\ \lceil (p(\pi_k) - p(\pi_j))/p(\pi_j) \rceil\}\}$$

$$\text{where } \hat{n}_t(\pi_k, \pi_j) = \sum_{\theta_{\pi_k,r} \in \Theta_{\pi_k}\ :\ \vartheta(\theta_{\pi_k,r}) \neq \vartheta(\theta_{\pi_j, n_g(\pi_j)})} n_t(\theta_{\pi_k,r})$$

By means of Lemma 1, we now arrive at a new sufficient, but not necessary, schedulability test that is based on the assumption that the EDF* policy is employed at run-time. The proof can be found in [15].

Theorem 2 (Jonsson, Lönn, and Shin). *In the presence of non-real-time threads, a set of real-time threads $\mathcal{T} = \{\tau_i : 1 \leq i \leq n\}$ with $d_i = p_i$ are schedulable on a uniprocessor with a primary-memory context-switch cost μ_g under the non-preemptive EDF* policy if the following conditions hold:*

1. $\sum_{\tau_i \in \mathcal{T}} c_i/p_i + \sum_{\pi_k \in \Pi} n_c(\pi_k) \times \mu_g / p(\pi_k) \leq 1.$
2. $\forall t \in S,\ \sum_{\tau_i \in \mathcal{T}} \lceil (t - p_i)/p_i \rceil^+ c_i + \sum_{\pi_k \in \Pi} \lceil (t - p(\pi_k))/p(\pi_k) \rceil^+ n_c(\pi_k) \times \mu_g + c_p \leq t,$

   where $S = \cup_{\tau_i \in \mathcal{T}} \{t : t \leq t_{max},\ t = p_i + mp_i,\ m \in \mathcal{N}\}$,

   $t_{max} = \max\{p_1, \ldots, p_n\}$

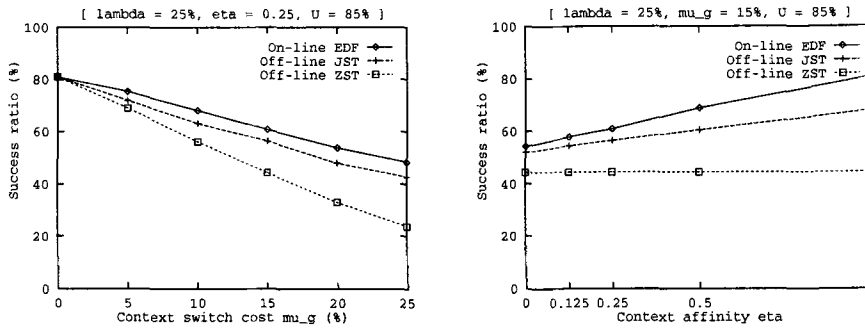

Fig. 4. Success ratio for dynamic-priority-driven dispatching as a function of a) the primary-memory context-switch cost μ_g assuming a context affinity $\eta = 0.25$; b) the context affinity η assuming a primary-memory context-switch cost $\mu_g = 15\%$.

Returning to the CLASSIFY algorithm in Fig. 3, we observe that the strategy used for enumerating the context groups within each period class is not specified in Step 3 of the algorithm. Recall that one of the strengths with the schedulability test in Theorem 2 is that it accounts for context affinity. More specifically, we observed that, given a certain period class π_k, only the context of the completion thread in period class π_j has to be checked against the threads in π_k to find the number of context switches caused by the completion thread returning from π_j to π_k. Now, in order to reduce the number of context switches, we must take advantage of the fact that only the completion thread has to be checked. This can be done by devising a good heuristic for enumerating the context classes within each period class. To this end, we make the following conjecture.

Conjecture 1. Assuming that the non-preemptive EDF* policy and the CLASSIFY algorithm are used, a good strategy for minimizing $n_c(\pi_k)$ is to guarantee that the completion thread is part of the context group that minimizes the following expression:

$$\theta_{\pi_k,r} \in \Theta_{\pi_k} \; : \sum_{\pi_j \in \Pi,\, \theta_{\pi_j,q} \in \Theta_{\pi_j} \,:\, p(\pi_j) > p(\pi_k) \,\wedge\, \vartheta(\theta_{\pi_j,q}) \neq \vartheta(\theta_{\pi_k,r})} n_t(\theta_{\pi_j,r})/p(\pi_j)$$

Fig. 4a plots the performance of the on-line EDF* algorithm as a function of the primary-memory context-switch cost μ_g. Also included in the plots is the modified Zheng/Shin schedulability test from Theorem 1 (denoted by ZST). The plots indicate that the EDF* policy performs much better than what can be predicted by ZST. For example, the performance of the on-line EDF* algorithm is more than 20 percentage points higher than the predicted one for $\mu_g \geq 20\%$. The reason for this poor correspondence between predicted and real performance is that ZST always assumes that a full primary-memory context switch will occur at a thread invocation. Since this strategy does not account for the variations in context-switch cost that can occur in an MLC architecture, the system will be poorly utilized if the context-switch cost is significant. In contrast, we can see that the predicted performance of the new schedulability test from Theorem 2

(denoted by JST) is significantly higher than that of ZST. In fact, the increase in predicted performance can be as high as 20 percentage points in systems with significant context-switch cost. This increase should, of course, be attributed to the "intelligence" that was added through the EDF* policy, the CLASSIFY algorithm and Conjecture 1.

Looking at the scheduling performance as a function of the context affinity, we observe in Fig. 4b that the performance of JST follows the curve for EDF* and increases for higher values of context affinity. This shows that the combination of the CLASSIFY algorithm and Conjecture 1 works very well. No such behavior can be noticed for ZST, since it does not account for the context affinity.

5 Related work and discussion

Little has been reported in the literature on the problem of real-time scheduling on MLC architectures, despite its growing importance. Humphrey *et al.* [5] recognize the need to exploit fast context-switch operations in order to improve the performance in the Spring kernel but did not make any attempt to evaluate the impact of such a technique on the application schedulability. Fiske and Dally [9] evaluated the use of a priority-based thread selection mechanism in a multithreaded architecture with multiple on-chip register sets. However, their work only evaluated scheduling strategies whose objective is to minimize the average application completion time. The work that comes closest to ours is that by Cheng *et al.* [13]. They proposed the dynamic-deadline-modification (DDM) heuristic for a multiprocessor MLC architecture assuming arbitrary thread preemption. As shown thus far, our proposed EDS heuristic clearly outperforms the uniprocessor version of DDM with respect to all the parameters considered.

The proposed strategies for reducing the number of imposed context switches during schedulability analysis is just a first attempt to address the problem of scheduling on MLC architectures. One interesting topic for future work is to investigate whether it is possible to find good heuristics for enumerating the context groups, and whether there exists an optimal enumeration strategy that minimizes the total number of primary-memory context switches at run-time. Such a strategy would have to consider not only period times and contexts but also execution times and phasings of threads.

6 Conclusions

In this paper, we have proposed and evaluated scheduling heuristics suitable for multi-level-context (MLC) architectures, where the context-switch cost is a function of the thread invocation pattern. Our main contributions are: (i) an integrated off-line scheduling heuristic that accounts for both the time-criticality of a thread and the current context affinity, and a demonstration of how this heuristic deliver superior performance over traditionally-used scheduling heuristics; (ii) an improved schedulability test for the on-line EDF algorithm that accounts for the variations in context affinity that can occur in an MLC architecture.

References

1. C. A. Healy, D. B. Whalley, and M. G. Harmon, "Integrating the Timing Analysis of Pipelining and Instruction Caching," *Proc. of the IEEE Real-Time Systems Symposium*, Pisa, Italy, Dec. 5–7, 1995, pp. 288–297.
2. R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon, "Timing Analysis for Data Caches and Set-Associative Caches," *Proc. of the IEEE Real-Time Technology and Applications Symposium*, Montreal, Canada, June 9–11, 1997, pp. 192–202.
3. C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the Association for Computing Machinery*, vol. 20, no. 1, pp. 46–61, Jan. 1973.
4. D. I. Katcher, H. Arakawa, and J. K. Strosnider, "Engineering and Analysis of Fixed Priority Schedulers," *IEEE Trans. on Software Engineering*, vol. 19, no. 9, pp. 920–934, Sept. 1993.
5. M. Humphrey, G. Wallace, and J. A. Stankovic, "Kernel-Level Threads for Dynamic, Hard Real-Time Environments," *Proc. of the IEEE Real-Time Systems Symposium*, Pisa, Italy, Dec. 5–7, 1995, pp. 38–48.
6. S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous Multithreading: A Platform for Next-Generation Processors," *IEEE Micro*, vol. 17, no. 5, pp. 12–19, Sept./Oct. 1997.
7. K. Diefendorff and P. K. Dubey, "How Multimedia Workloads will Change Processor Design," *IEEE Computer*, vol. 30, no. 9, pp. 43–45, Sept. 1997.
8. A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin, "Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors," *IEEE Micro*, vol. 13, no. 3, pp. 48–61, June 1993.
9. S. Fiske and W. J. Dally, "Thread Prioritization: A Thread Scheduling Mechanism for Multiple-Context Parallel Processors," *Proc. of the IEEE Symposium on High Performance Computer Architecture*, Raleigh, North Carolina, Jan. 22–25, 1995, pp. 210–221.
10. C. A. Waldspurger and W. E. Weihl, "Register Relocation: Flexible Contexts for Multithreading," *Proc. of the ACM Int'l Symposium on Computer Architecture*, San Diego, California, May 16–19, 1993, pp. 120–130.
11. J. Jonsson, "GAST: A Flexible and Extensible Tool for Evaluating Multiprocessor Assignment and Scheduling Techniques," *Proc. of the Int'l Conf. on Parallel Processing*, Minneapolis, Minnesota, Aug. 10–14, 1998, pp. 441–450.
12. C.-Y. Lee, J.-J. Hwang, Y.-C. Chow, and F. D. Anger, "Multiprocessor Scheduling with Interprocessor Communication Delays," *Operations Research Letters*, vol. 7, no. 3, pp. 141–147, June 1988.
13. B.-C. Cheng, A. D. Stoyenko, T. J. Marlowe, and S. Baruah, "A Scheduler Maximizing Maximum Tardiness for DSP Programs with Context Switch Overheads Considered," *Proc. of the Int'l Conf. on Signal Processing Applications & Technology*, Boston, Massachusetts, Oct. 7–10, 1996, pp. 771–775.
14. Q. Zheng and K. G. Shin, "On the Ability of Establishing Real-Time Channels in Point-to-Point Packet-Switched Networks," *IEEE Trans. on Communications*, vol. 42, no. 2/3/4, pp. 1096–1105, Feb./Mar./Apr. 1994.
15. J. Jonsson, H. Lönn, and K. G. Shin, "Non-Preemptive Scheduling of Real-Time Threads on Multi-Level-Context Architectures," Technical Report No. 98–6, Dept. of Computer Engineering, Chalmers University of Technology, S–412 96 Göteborg, Sweden, May 1998.