Parallel Rendering of Human-Computer Interaction Industrial Applications on Multi-/Many-Core Platforms

Sven Hermann, Arquimedes Canedo, and Lingyun (Max) Wang

Siemens Corporation, Corporate Technology
Princeton, NJ, USA
{sven.hermann,arquimedes.canedo,max.wang}@siemens.com

Abstract. Industrial Human Computer Interaction (Industrial HCI) devices are beginning the transition from single-core to multi-/many-core technology. In practice, improving the real-time response time of graphical user interface (GUI) applications in multi-/many-core is difficult. This paper presents a novel parallel rendering approach targeted to improve the performance of Industrial HCI applications in multi-/many-core technology. This is accomplished through the identification of coarse-grain parallelism during the application design, and the exploitation of fine-grain parallelism during runtime using a dynamic scheduling algorithm and true parallel execution of GUI workloads. Using a real benchmark application, we show that response time can be reduce by up to 217% in a quad-core processor.

1 Introduction

Industrial HCI (Human Computer Interaction) devices are real-time embedded computer systems, based on Graphical User Interface (GUI) applications, which allow humans to interact with and control complex industrial processes such as power plants, manufacturing lines, chemical processes, and transportation systems. The performance gap between high-end and low-end Industrial HCIs is quite substantial, and this causes additional design, development, manufacturing, and maintenance costs for Industrial HCI manufacturers. For example, multimedia and video processing in Industrial HCIs requires high-performance CPUs and GPUs, while basic input/output processing requires low-power embedded processors. The technological shift from single-core processors to multi-/many-core processors is very attractive for Industrial HCI vendors because the performance gap in multiple products can be eliminated by consolidating a line of Industrial HCIs with the same multi-/many-core processor rather than having different custom processors for different product configurations. While it is clear that multi-/many-core processors provide better performance, energy efficiency, scalability, consolidation, and redundancy than single-core processors, it is still an open question how to best utilize the additional cores for improving the performance and response time of GUI applications.

M. Kurosu (Ed.): Human-Computer Interaction, Part I, HCII 2013, LNCS 8004, pp. 350–360, 2013.

This paper presents a novel "**parallel rendering approach for GUI applications**", defined as the process by which an image is generated **cooperatively** and **concurrently** by independent computation threads running on different cores. **Our approach aims at accelerating the response time of Industrial HCI Devices through parallel execution of GUI-related workload in multi-/many-core processors**. Finding parallelism at the GUI object level is challenging and we identify three steps that are necessary to expose and exploit it. First, the application is analyzed for object dependencies using data flow analysis in a process we refer to as dependency-based load balancing where independent clusters of GUI objects are scheduled into different cores. Second, the worker threads execute workload in parallel and are allowed to modify the objects' data directly. Third, after the worker threads have finished, or a display update event is received, the "flush thread" optimizes the sequential access to the display by minimizing the number of pixels and the number of draw function calls. Our original contributions are:

- A method for parallel rendering of GUI applications in multi-/many-core based Industrial HCI devices.
- The extraction of object-level parallelism based on a dependency-based load balancing algorithm.
- The execution of GUI objects' in parallel through privatized memory.
- The reduction of display access through a flush call optimization.
- An implementation of our method on a quad-core system and its evaluation using an Industrial HCI benchmark.

The rest of this paper is organized as follows. Section 2 describes the limitations of the current Industrial HCIs and motivates the need for multi-/many-core-based Industrial HCIs. Section 3 presents the parallel rendering method including the load balancing, parallel execution, and flush optimization algorithms. Section 4 presents our experimental results on a soft real-time quad- core-based Industrial HCI. Section 5 concludes the paper and sets the direction for future work.

2 Industrial HCIs – A Review of the State-of-the-Art

Industrial HCI applications have two phases: the engineering phase, and the runtime phase. The *engineering* refers to the design of the screen and the definition of its functionality. For example, buttons to trigger certain actions, image display, drawings, status bars, file system menus, communication with Programmable Logic Controllers (PLC), etc. The *runtime*, on the other hand, refers to the actual execution of these programs in an embedded computer system and it is necessary for users to interact with the GUI. In this Section, we present the state-of-the-art in engineering and runtime implementations of Industrial HCIs to motivate and highlight the need for parallel computing.

2.1 Engineering System

Industrial HCI vendors hide the complexity of the underlying architecture to the Industrial HCI application designer for various reasons. First, a single engineering system is used for multiple Industrial HCI devices with different capabilities and this level of abstraction allows the same application to have the same look and feel in all the Industrial HCIs. Second, the job of the designer should be focused on dealing with GUI objects and their associated actions and attributes, and not on dealing with the underlying computer system. **The programming abstractions introduced by the current engineering systems, unfortunately, are not suitable for the next generation of computation elements because they assume that the underlying computation element is always a single-core processor**. Embedded processor manufacturers are moving towards multi-/many-core technology [1–3] and we expect the next-generation Industrial HCI devices to adopt parallel processing technology. Typically, Industrial HCI screens consist of several objects positioned and configured by the application developer. Each object may include a list of Actions that, on every cycle or when an event occurs, the Industrial HCI runtime will execute. These Actions range from changing values (e.g. SetValue) of variables (Tags), to changing the appearance and properties of the objects themselves (e.g. color, X-position, Y-position). In a single-core implementation, the runtime system executes all the objects' Actions, one by one. Clearly, this approach does not scale anymore as single-core processors have reached a speed plateau. In a multi-/many-core system, the runtime can implicitly exploit parallelism by assigning these Actions to different threads and scheduling these into different cores. However, it is critical that the parallelization is done carefully in order to obtain performance benefits from multi-/many-core. Poor parallelization of multi-threaded programs often leads to performance degradations when compared to single core due to excessive synchronization and communication overhead.

2.2 Runtime

The runtime is responsible for executing the Industrial HCI application created with the engineering system in a timely manner in order to comply with the real-time requirements. Figure 1 shows the conventional implementation of runtime systems for GUI applications. GUI applications are driven by user triggered events such as a mouse click and/or system events such as timer and alarms. These events are detected by a single "GUI thread" responsible for handling all events and managing all the GUI-related objects and operations such as updating the display. To maintain the application response time as short as possible, the GUI thread runs under an infinite loop that detects and dispatches events to the event handling functions allocated to worker threads that perform the handling of the event. Whenever the GUI thread is not processing events in a timely manner, the users may experience an unresponsive application that "freezes". Although this model decouples event detection from event handling

in multiple worker threads, it does not scale well in multi-/many-core processors because these worker threads must use the GUI thread to modify GUI objects' data and the display as shown in Figure 1. **Unfortunately, this creates a serialization bottleneck that eliminates any possibilities of improving the response time with multi-/many-core because all the GUI related workload is concentrated on a single thread**. The main observation is that the existing runtime model suffers from a performance bottleneck when executed in multi-/many-core processors because worker threads are not allowed to modify GUI objects directly. Instead, the GUI thread is responsible for all GUI-related data and this inhibits scalability in modern multi-/many-core processors.Worker threads have to enqueue 'Update object Events' back to the GUI thread which then will handle all the GUI object related workload, e.g. updating the color change of a pressed button, in a serialized manner.

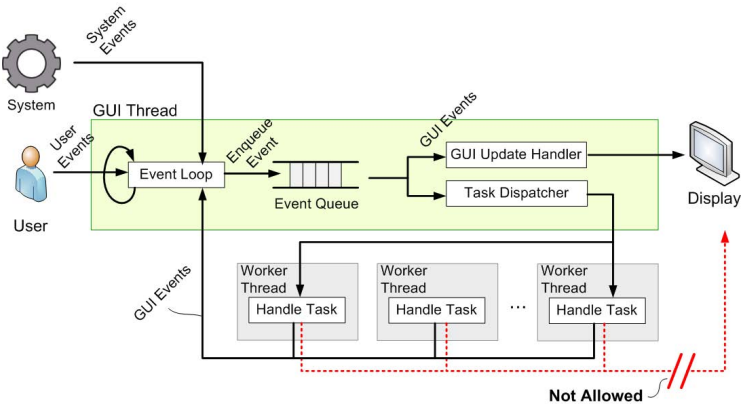

Fig. 1. The existing GUI application model suffers from a performance bottleneck in multi-/many-core processors because worker threads are not allowed to modify GUI objects

Current Industrial HCI runtime systems are also affected by their limited ability to transfer GUI objects to the screen after all the tasks are executed and the data has been updated by the worker threads. This transfer, referred to as *flushing*, can be performed one object at a time, or bundling several objects into a single flush call. Both approaches have advantages and limitations. Flushing single objects is simpler and faster but the number of flush calls can be a performance penalty in some systems. Bundling multiple objects and flushing once reduces the number of flush calls but increases the memory bandwidth requirements. Unfortunately, the flushing strategy greatly depends on the underlying hardware configuration. Current Industrial HCI runtime systems often perform the flush operations based on the objects' bounding boxes and clustering them according to the order by which they were created during the engineering phase. For example, existing runtime systems would group the 12

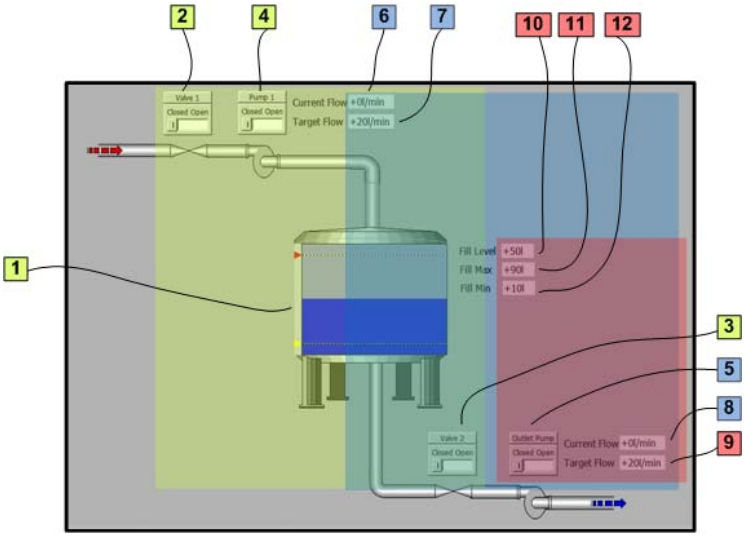

Fig. 2. Naive clustering of GUI objects based on their creation order incurs in area overlaps that generate additional unnecessary work

GUI objects in a large capacity water tank control system shown in Figure 2 into three sets $< 1, 2, 3, 4 >$, $< 5, 6, 7, 8 >$, $< 9, 10, 11, 12 >$ in order to perform three flush operations. Notice that the objects are clustered according to their creation index. Although clustering multiple objects reduces the total number of calls, it incurs in additional overhead related to calculating the size of the aggregated bounding box that encloses the objects in a set. Also, notice that there may be clusters that overlap in space and this creates unnecessary and redundant flushing. This shows that a redesign of the runtime system for Industrial HCIs is also necessary for the adoption of multi-/many-core technology. In this paper, we focus on two key aspects of the design: effective multi-threading in multi-/many-core, and optimization of the flushing operations.

2.3 Related Work

Our work relates to the desktop application parallelization research. In [4], the authors present an object-oriented parallel programming library for GUI applications and a dynamic runtime system that allows the parallelization of image processing applications in multi-/many-core processors. Due to the streaming nature of multimedia applications, other researchers have demonstrated that vectorization [5, 6] and custom parallel hardware [7] are other effective means to accelerate desktop applications. The common aspect to all the related work is the focus on the parallelization of non-real-time multimedia applications that are known to take advantage of parallel processing [8]. Our work, on the other hand, focuses on the parallelization of real-time sub-millisecond applications with tight dependencies between user-actions and data processing.

3 Parallel Rendering for Industrial HCIs

In order to improve the performance, energy efficiency, scalability, consolidation, and redundancy in Industrial HCIs, we propose a novel parallel rendering technology that effectively uses multi-/many-core processor technology to reduce the response time of GUI applications bound to real-time requirements. To overcome the sequential computation limitations inherent to the current Industrial HCI programming (See Section 2), we propose a parallel rendering method in both the engineering system and the runtime system to exploit parallelism in these applications.

3.1 Identifying Coarse-Grain Parallelism in the Engineering System

In our system, parallelization may begin at the engineering system. The GUI designer is often the best person to identify coarse-grain parallelism opportunities because the layout and functionality of an Industrial HCI screen is closely related to the underlying industrial automation pyramid [9] consisting of sensors, actuators, controllers, SCADA (Supervisory Control and Data Acquisition), MES (Manufacturing Execution Systems), and ERP (Enterprise Resource Planning) systems. We strongly believe that this intuition provides an excellent opportunity for our system to expose an initial coarse-grain concurrency. Figure 3 shows the process of identifying coarse-grain parallelism at the engineering system. The *execution group* (EG) selection step is introduced to the GUI design process to bind the Actions in the objects on the screen to *suggested* logical threads. It must be noted, however, that these are simply hints provided by the Industrial HCI designer and the ultimate execution of Actions in specific cores is up to the runtime scheduling algorithms. In addition to the list of available EGs, the "Default" setting is the default assignment for Actions and it implies that the designer is unsure about the assignment and it is completely up to the runtime to decide how Actions are executed in the available cores in the system.

3.2 Exploiting Fine-Grain Parallelism in the Runtime System

Although our engineering system exposes coarse-grain parallelism, this parallelism is still subject to the existing runtime system limitations discussed in Section 2.2. Even though multiple threads exist in the application, these are not allowed to modify GUI objects' data directly. Thus, a serialization bottleneck prevents threads to take advantage of multi-/many-core processors. To eliminate this serialization bottleneck, we propose a runtime system that allows different threads to access a privatized memory area and this allows the application to truly execute GUI applications in parallel. The responsibility of our runtime system is to schedule the execution of Actions of the objects to different CPU in such a way that the response time of the Industrial HCI application is reduced.

Figure 4 highlights the three main differences between our parallel rendering runtime system and conventional runtime systems. Although our approach also uses a GUI thread to enqueue input events (e.g. user inputs, system events,

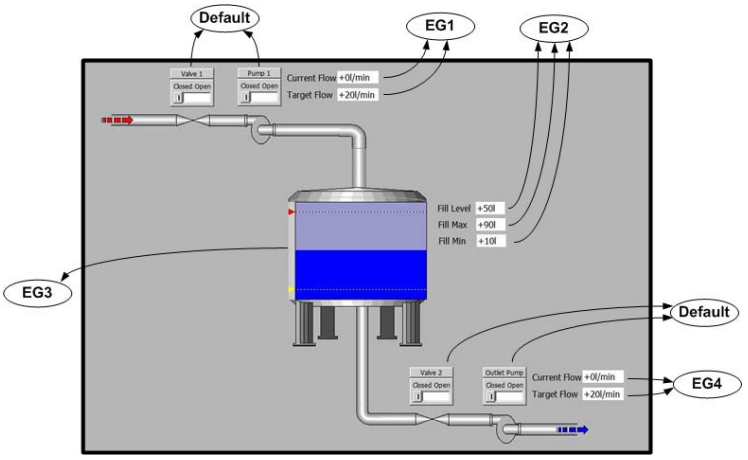

Fig. 3. Identifying coarse grain parallelism through EG selection during application engineering

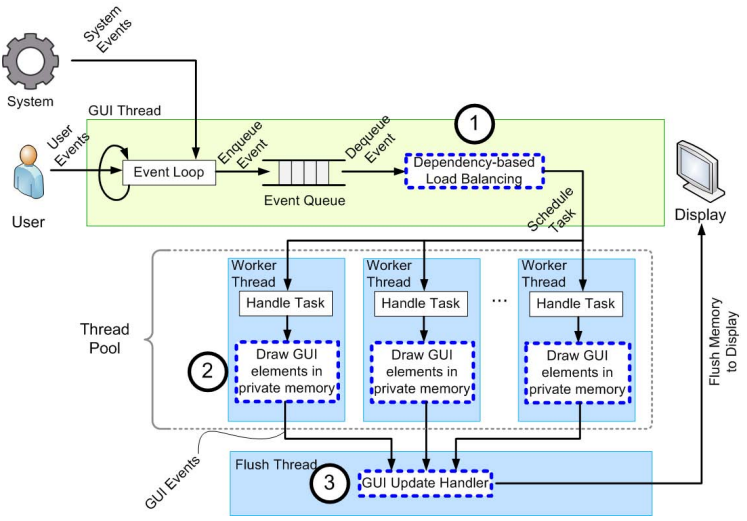

Fig. 4. Our parallel rendering method eliminates the serialization bottleneck in the GUI thread by ① performing a dependency analysis at the GUI object-level and distributing the workload to multiple worker threads,② allowing the worker threads to modify GUI objects directly, and ③ optimizing the flush of the internal memory to the display

interrupts, etc.), the first difference is that this thread is now also responsible for performing a dependency-based load balancing ① to distribute the workload to multiple worker threads. Second, the worker threads now use a privatized memory area ② to truly parallelize the execution of objects' Actions and eliminate the serialization bottleneck created in the conventional systems. Third,

a dedicated flush thread ③ optimizes the data transfer between the worker threads private memory and the display in order to reduce the data size and the function call frequency.

The dependency-based load balancing is the first critical step for exploiting parallelism in Industrial HCI applications running on multi-/many-core. The key observation about the dependency-based load balancing algorithm is that it uses runtime information, in addition to the static coarse-grain information provided by the engineering system, to determine the data dependencies between GUI objects. As shown in Figure 5, this is accomplished through a dynamic data dependency analysis that groups the dependent GUI objects into clusters that are dispatched to different cores.

Maintaining data dependent objects in the same core minimizes synchronization and communication among cores because worker threads only access their private memory area and this ultimately helps to reduce the response time. The private memory area mechanism guarantees that only one CPU accesses that area, and in combination with the data dependency analysis, it enables the possibility of parallel execution of GUI applications as shown by ② in Figure 6.

Since the worker threads now run under different time constraints, caused by uneven workloads of each GUI object, it is necessary to synchronize them after their execution cycle completes. As shown in Figure 6, the "flush thread" ③ optimizes the sequential access to the display by minimizing the number of pixels and the number of draw function calls in order to reach the maximum performance gain when transferring the GUI data to the display. The fundamental optimization mechanism is to minimize both the size of the bounding box of multiple GUI objects and the number of calls necessary to flush them while avoiding the creation of overlapping bounding boxes. For example, Figure 6 shows that the algorithm determines that 4 non-overlapping clusters consisting of elements$< 1, 3, 8, 6 >$, $< 7, 4 >$, $< 2, 9 >$, and $< 5 >$ is the optimal strategy for flushing the screen in a particular system. It is important to note that the optimal balance between number of calls and size is highly influenced by the underlying hardware configuration. Nevertheless, compared to existing approaches, our method effectively eliminates the overlap regions and therefore unnecessary workload, and also reduces the size resulting in shorter flush times.

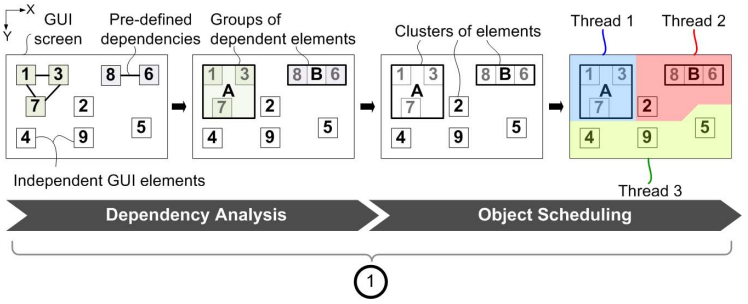

Fig. 5. Dependency-based object scheduling ①

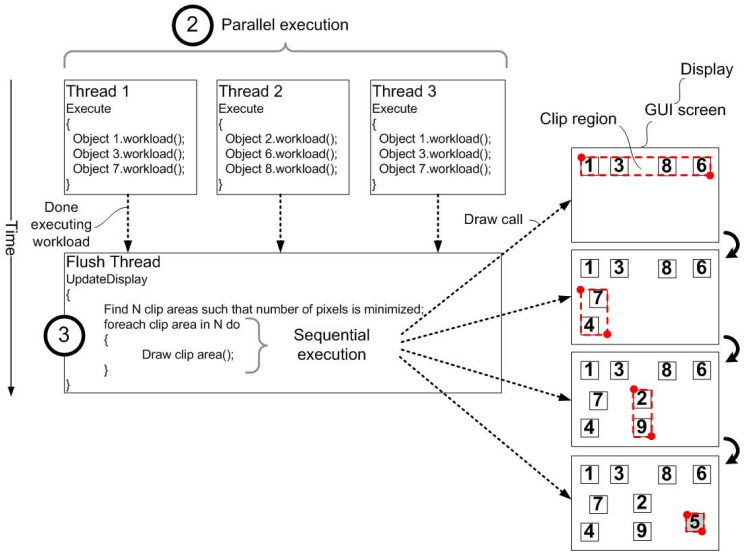

Fig. 6. Parallel workload execution ② and flushing optimization ③

4 Experimental Results

To validate our concepts, we implemented a parallel rendering system for Industrial HCIs in a commercial-off-the-shelf quad-core processor. Using a realistic benchmark used to test the response time of existing runtime systems, we focus on characterizing our parallel rendering system in terms of two key design aspects: the scalability of multi-/many-core based Industrial HCI systems against single-core implementations, and the effects of multi-/many-core scheduling in the performance of GUI applications.

Figure 7 shows the rendering time of six configurations of the benchmark when executed in a single and four cores. Our parallel rendering framework is capable of reducing the rendering time on five of the six configurations from 36% to 217%. The "1 Rectangle" configuration shows a performance degradation of -3%. These results support our main objective of providing faster response time on Industrial HCI applications by parallel execution of μ-second-level workloads on multi-/many-core processors.

Figure 8 compares the speedup factors relative to single-core execution obtained by a conventional runtime and our parallel rendering runtime. The difference is that our method exploits fine-grain parallelism in multiple-cores. Notice that using a conventional straightforward scheduling, only one out of six configurations benefit from parallel execution while the other five represent performance degradation of up to -58%. This performance degradation is due to the fact that the current runtime systems are incapable of effectively exploiting parallelism and a simple scheduling policy is not sufficient for taking advantage of

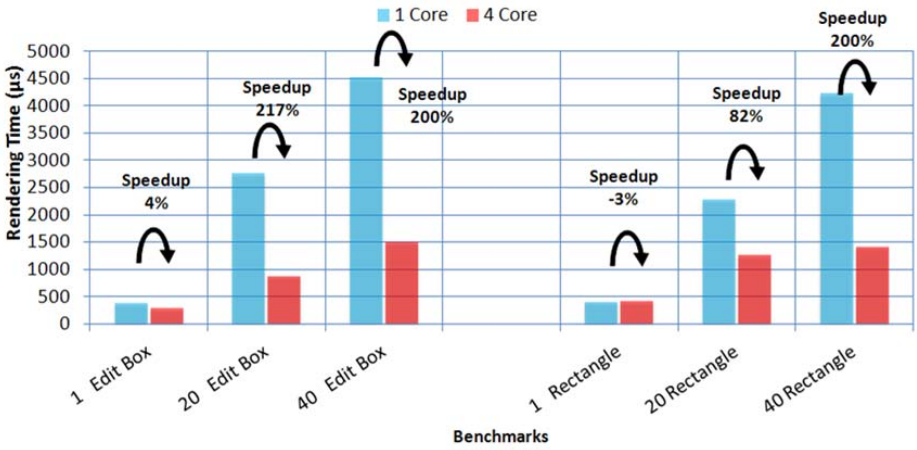

Fig. 7. Sub-millisecond level workloads can be reduced by up to 217% when executed by our method in a quad-core processor

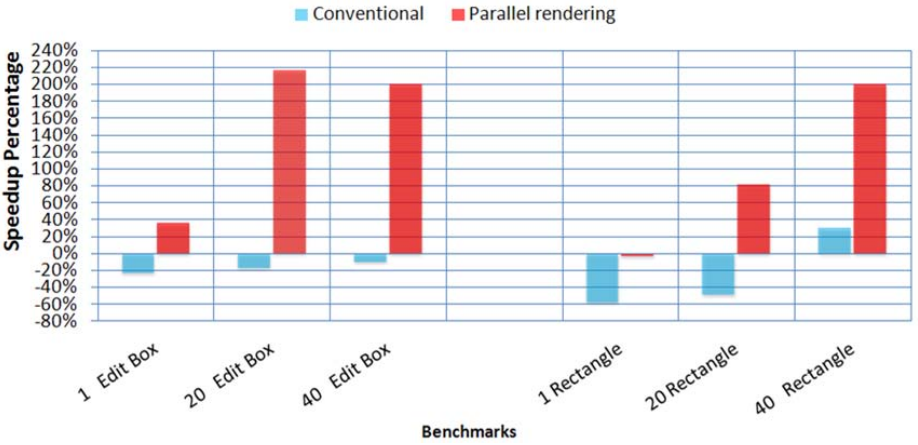

Fig. 8. Speedup percentages of existing runtimes and our parallel rendering method relative to execution on single core

multi-/many-core Industrial HCIs. The parallel rendering method, on the other hand, shows improvements of up to 217%.

5 Conclusion

To facilitate the transition from single-core to multi-/many-core Industrial HCIs, we developed a parallel rendering method to eliminate the serialization bottleneck that exists in state-of-the-art engineering and runtime systems. Our

algorithm uses a static parallelization method at the engineering stage to identify coarse-grain opportunities. This information is propagated to the runtime stage where additional dynamic information is used to exploit fine-grain parallelism. This is assisted by a dependency analysis, object scheduling, parallel processing of GUI elements, and non-overlap flushing algorithms that are necessary to guarantee that real-time Industrial HCI applications are executed faster in multi-/many-core. Our experiments show that our implementation is capable of reducing the response time of a real Industrial HCI benchmark by up to 217%. In our future work, we plan to extend our method to process user and system-level events in parallel. This approach would be particularly useful for the acceleration of industrial HCIs with modern multi-touch interfaces [10].

References

1. Intel: Technical Resources for Embedded Designs with Intel Architecture, `http://www.intel.com/`
2. Freescale: QorIQ Processing Platforms - Industrial, `http://www.freescale.com`
3. Texas Instruments: Stellaris MCU for industrial automation, `www.ti.com`
4. Giacaman, N., Sinnen, O.: Object-Oriented Parallelization of Java Desktop Programs. IEEE Software (1), 32–38 (2011)
5. Luk, C.K., Newton, R., Hasenplaugh, W., Hampton, M., Lowney, G.: A Synergetic Approach to Throughput Computing on x86-Based Multicore Desktops. IEEE Software (1), 39–50 (2011)
6. Pankratius, V., Schulte, W., Keutzer, K.: Parallelism on the Desktop. IEEE Software: Guest Editors' Introduction (1), 14–16 (2011)
7. Draper, B., Beveridge, J., Bohm, A., Ross, C., Chawathe, M.: Accelerated image processing on FPGAs. IEEE Transactions on Image Processing 12(12), 1543–1551 (2003)
8. Blake, G., Dreslinski, R.G., Mudge, T., Flautner, K.: Evolution of Thread-Level Parallelism in Desktop Applications. In: International Symposium on Computer Architecture, ISCA (2010)
9. Nof, S.Y.: Handbook of Automation. Springer (2009)
10. Beaudouin-Lafon, M.: Instrumental interaction: an interaction model for designing post-wimp user interfaces. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 2000, pp. 446–453 (2000)