JEOPARD – Java Environment for Parallel Real-Time Development

(Invited Paper)

Fridtjof Siebert
aicas GmbH
Karlsruhe, Germany
siebert@aicas.com

Abstract

Multicore systems have become standard for desktop computers today. Current operating systems and software development tools provide straightforward means to use the additional computing power. However, a more fundamental change in the design and development of software is required to fully exploit the power of multicore systems. Furthermore, the fast growing market of embedded systems is currently largely unaffected by the introduction of multicore systems. This will change quickly in the future, which will mean that there will be a demand on efficient development of reliable embedded software that can give real-time guarantees and exploit the available power on multicore systems.

The JEOPARD project addresses this demand by developing Java software tools to exploit multicore power while ensuring correctness and predictable timing. This paper gives an overview of the JEOPARD project and focuses on key technical issues such as real-time scheduling and real-time garbage collection on multi-core systems.

1. Introduction

It is recognised that there is a new software crisis facing the industry. To continue the increasing growth of computer power requires a transition to multicore architectures, yet our software development models are rooted in sequential programming. Add into this mix the growing use of embedded computers and their ever increasing demand for better performance within constrained power consumption (possibly battery supplied). The result is an urgent requirement for tools to support multicore embedded software development.

Whilst it is currently beyond the state of the art to productively develop software for the full power provided by multicore architectures [1], the JEOPARD project believes that there is a migration path that can be plotted from the current state of embedded systems practice to a multicore future.

Multicore symmetric multiprocessor systems (SMP) have become standard for desktop computers today and current operating systems and software development tools provide means to use this additional computing power. Increasingly, embedded systems are being proposed and built using multicore systems. Future embedded systems may well exploit more complex multicore architectures such as NUMA (Non-Uniform Memory Architecture) and Network-on-Chip (NoC).

In the near future, higher demands will be made on these embedded systems and greater reliance will be placed on the delivery of their services. More and more of these systems will become high-integrity systems whose failure can cause loss of life, environmental harm, or significant financial loss.

The increasing demand on the processing power for the more and more complex tasks realised by these systems can only be met by the use of multicore systems. In addition to desktop systems, embedded systems have requirements on predictable timing behaviour and safety-criticality.

Hence, the JEOPARD project believes that the transition to full multicore platforms can be achieved in two stages. The first stage is to provide an environment within which embedded software can be developed for SMP platforms. Embedded systems use a large variety of different architectures, such that portability via uniform interfaces to the systems resources is required to support the versatility in terms of performance, power and coping with devices that range from low-end controller systems to powerful high-end embedded systems. Whilst many of these systems can be reflected in an SMP-type of architecture, others require a NUMA. Integrating access to these architectures within a software development environment now requires study and the development of prototype tools. These tools need to be evaluated to determine to what extend they can be used for future industrialisation. This is the second stage of the transition. The study and prototyping can be done in parallel with the first stage.

The main strategic objective of the JEOPARD project is to provide the tools for platform-independent development of predictable systems that make use of SMP multicore platforms. These tools will enhance the software productivity and reusability by extending technology that is established on desktop system by the specific needs of multicore embedded systems. The project will actively contribute to standards required for the development of software in this domain.

Whilst current platforms may consists of between 4 and 8 homogeneous RISC processors with an SMP architecture style, this approach does not scale to future platforms where a NUMA architecture is more appropriate. Consequently, the JEOPARD project will undertake research into how such systems can be programmed and analysed for their real-time properties. Where appropriate proof-of-concept prototypes will be developed.

Hence, the JEOPARD project attempts to strike a balance between meeting the need for industrial strength tools to ameliorate the current crisis, and plotting a migration path to future exploitation of the full power and generality of multicore platforms.

The JEOPARD consortium consist of the following industrial, academic, technology and standardisation organisations: aicas GmbH (Germany), EADS Deutschland GmbH (Germany), FZI Karlsruhe (Germany), RadioLabs (Italy), SkySoft (Portugal), Sysgo (France), Technical University Cluj-Napoca (Romania), Technical University of Vienna (Austria), and the University of York (UK).

2. Multicore Technologies

The JEOPARD project has partners that work on all layers involved in a multicore system, from the bare hardware within a multicore processor, via a multicore real-time OS, a multicore real-time Java VM, Java APIs for multicore systems up to different critical applications and tools for the analysis of the correctness of these applications. The following sections will explain these individual layers in more detail.

An overview of the different building blocks for the JEOPARD toolchain is shown in Figure 1. Central to the toolchain is the Java application. The developer can use specifications developed by the JEOPARD project partners to make use of multiple processors. The application is run on two possible execution environments, a VM running on off-the-shelf multicore processors, or a parallel Java bytecode processor. For the correctness validation, a static analysis tool and a concurrent unit testing tool are developed.

2.1. CPU Architecture Layer

2.1.1. Architecture Alternatives. The basis of a multicore architecture is a Chip Multiprocessor (CMP). The JEOPARD project will consider three basic variants of a CMP to support Java programs targeted at multicore platforms:

1) SMP – a symmetric arrangement of conventional processors, as typified by Pentium class multicore CPUs.
2) Multi-JOP – a number of hardware Java processors with some shared memory (based upon the Java processor JOP, [11], [12]). JOP is a small processor design and we will implement 8 (or more) cores on a medium sized field programmable gate array (FPGA).
3) NUMA – to incorporate future architectures, including those with substantial field-programmable elements.

For all architectures, the project will develop efficient support for higher level OS and Java based services. In particular, specific services for synchronisation, scheduling support and deterministic automatic memory management – these will improve overall system performance.

2.1.2. Aims. The scientific aims of the CPU architecture layer of JEOPARD are:

- to achieve a better understanding of the tradeoffs between software and hardware implementations of Java core functionality;
- to examine the bottlenecks inherent in current implementations of the Java VM within real-time systems;
- to examine and understand the implications of future multicore and CPU architecture trends upon software and hardware implementations of the Java VM, including presence of on-chip field-programmable elements.

2.1.3. FPGA support. The applications that will be developed by the JEOPARD user partners have computation intensive components, such that some algorithms shall be implemented directly in hardware and run in FPGAs parallel to the main Java application running on an multicore CPU platform.

Therefore, JEOPARD will design and implement an interface between the JOP Java processor or the JamaicaVM Java virtual machine (see 2.3) and Java methods implemented in hardware that execute on an FPGA. This interface will be based upon that developed within the Javamen project [3], which established a basic method call interface between a single JOP and hardware methods. This enabled acceleration of key Java methods within the application by their implementation as softcores within the FPGA. The Javamen approach will be expanded to enable a multicore JOP architecture (ie. multiple JOP processors) to interface to and share methods implemented in hardware on an FPGA.

2.2. OS Layer

2.2.1. Multicore RTOS. One programming model for a multicore CPU is that of an SMP, as used in many applications from supercomputers to high-performance server systems for many decades. Within the JEOPARD project, the OS layer is considered within an SMP context only (ie. the OS is not considered for the Java-CPU Multi-JOP, which has no need for an OS, nor within NUMA). In order to exploit the power of the platform, efficient and effective use of all CPUs must be made. This can be achieved in many ways, from explicit use of all CPUs, perhaps via a separate OS on each CPU; through to virtualising the platform such that higher levels are not aware of the underlying SMP platform.

The computational load of server systems typically consists of a large number of mostly independent processes. Moreover, the performance of a server system is usually

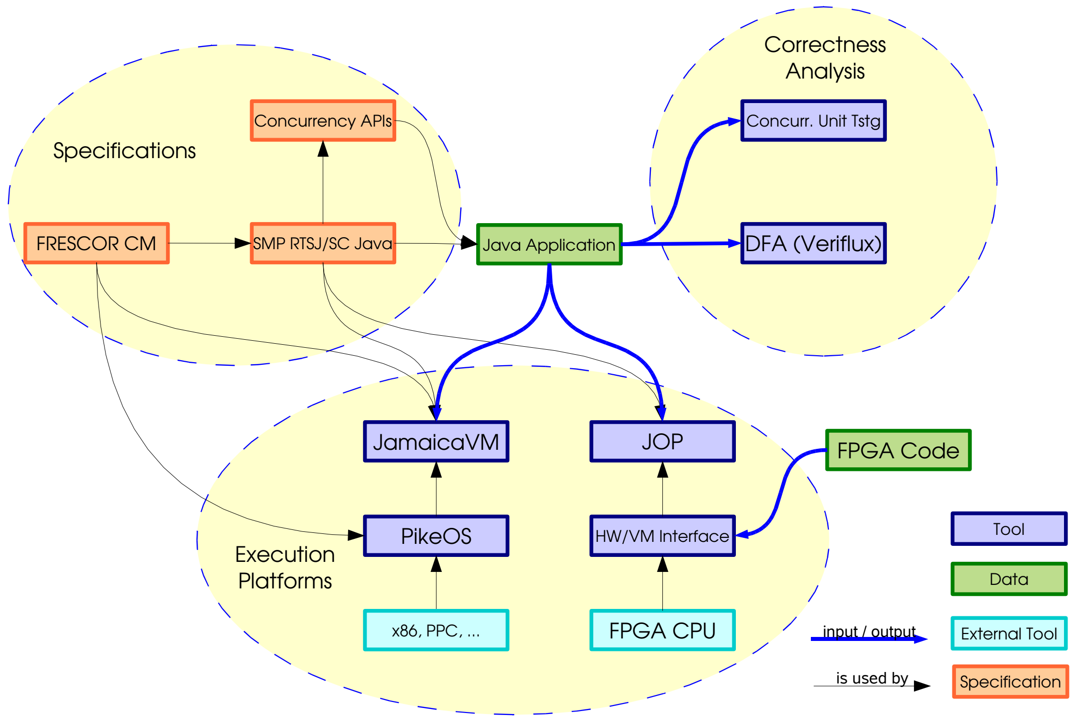

Figure 1. *The JEOPARD toolchain consists of three main aspects: specifications, correctness analysis and execution platforms.*

measured as average throughput: If there are any deadlines to be met by processes at all, they are “soft” at best, i.e. missing a deadline is not considered to be a fatal error. With such requirements, a “load balancing” approach is entirely suitable: all processes are treated alike at the operating system level. They are selected for execution on a “first come, first serve” basis as processor cores become available. Today, most multiprocessor operating systems work by such a strategy. In some cases, heuristics are used in addition, e.g. to improve cache utilisation by avoiding excessive process migrations. Obviously, such techniques are not suited for scheduling processes with hard timing requirements.

For a real-time system, these techniques are not sufficient, more important than high average throughput are worst-case execution times of each single task to ensure that all deadlines will be met and that priorities are respected.

2.2.2. Multi-core partitioning RTOS. A multi-core version of the partitioning RTOS PikeOS ([9], [10]) will be developed. This includes the definition and implementation of a system call interface that enables strict resource separation between multiple user-level domains. Resources to be considered are memory, I/O ports, kernel resources (i.e. memory consumed by the kernel on behalf of applications) and processor-time (i.e. allocation of a set of processor cores for a given amount of time). The separation of these resources is the necessary foundation of a multiprocessor MILS system. The user-level domains established in this way are independent virtual multiprocessor machines which are able to host individual real-time JVMs.

The API definition will be derived from the PikeOS microkernel’s API which already supports separation of all the resources listed above, except for processor-time. The API support for the FRESCOR project’s contract model (see 2.5.2) will form the starting point for processor-time partitioning.

2.3. VM Layer

For real-time Java applications to run on multicore embedded systems, a multicore real-time Java VM implementation is required. The runtime services of the VM such as the interpreter, thread synchronisation, memory management, etc. must be implemented to exploit the available processors while executing predictably and efficiently.

The implementation will be based on the uniprocessor real-time Java VM JamaicaVM [8]. All aspects that are currently limited to a single CPU will be reimplemented to enable parallel execution on multicore systems, while every attempt will be made to maintain the real-time behaviour.

2.3.1. Parallel Real-Time GC. As an example of a core technology developed by JEOPARD, section 3 will explain the difficulties encountered in a parallel GC and their solutions in JEOPARD in more detail.

2.3.2. Parallel Java Monitors. The parallel execution of thread synchronisation via Java monitors requires a careful implementation of the primitive monitor operations such as entering, exiting and notify. Also, the Java thread scheduler must know about multiple processors, parallel Java interpretation and JIT compilation.

2.3.3. Parallel Real-Time VM. The result of the VM layer work will be the first deterministic Java implementation for parallel systems. The users will be able to develop deterministic parallel applications while profiting from the safety, flexibility and productivity of a Java environment.

2.4. API Layer

For a standard desktop computing environment, the application programmer is content to allow the OS to manage the access to the computing resources. In an embedded environment this is generally not the case. Better real-time performance can often be obtained by partitioning the applications threads between the core processors. Currently, there is no easy way to do this using Java or the RTSJ API [2]. This is an accepted limitation of the current real-time Java technology. Although some consideration has been given to an API to do this within the context of JSR 282 [4], the proposal is not based on practical experience.

The work on the API layer is concerned with extending the Java programming model to allow access to the resources in a machine-independent way. The focus will be on providing support for static partitioning but dynamic aspects will also be considered.

The implementation of this extended functionality will build on appropriate support to be provided by the underlying OS layer. Specifically, the OS will have to enable the Java virtual machine to fully control parallel vs. time-multiplexed execution of its threads as well as to control which threads are executed on which processor core (i.e. processor affinity) and even to migrate threads between cores. Current state-of-the art RTOSes fail to provide this kind of functionality. Also, OS API standards do not yet specify such functionalities. Therefore, the JEOPARD project will develop new definitions and specifications of such new OS API functionalities.

In addition, the APIs to allow hardware-implemented FPGA components to be integrated within the overall Java framework will be addressed. Furthermore, the impact at the API-level of a more general NUMA architecture will be considered.

2.4.1. Real-Time for Multicore and Standards. The Real-Time Specification for Java (RTSJ) is an established standard for real-time software development in Java [2]. The JEOPARD project therefore works closely together with the RTSJ-related Java Specification Requests (JSR 282) to ensure that future versions of the RTSJs are conforming to the requirements that arise on multicore systems [15].

In order to make the RTSJ fully defined for SMP multiprocessor systems, the following issues need to be addressed.

1) The dispatching model – the current specification has a conceptual model which assumes a single run queue per priority level.
2) The allocation model – the current specification provides no mechanisms to support processor affinity,
3) The synchronisation model – the current specification does not distinguish between synchronised methods that suspend holding their locks and those that do not,
4) The cost enforcement model – the current specification does not consider the fact that processing groups can contain scheduling objects which might be simultaneously executing,
5) The affinity of interrupts (happenings) – the current specification provides no mechanism to tie interrupts (happenings) and their handlers to particular processors, and
6) The failure model – the current specification makes no statements about partial failures of the underlying platform.

The JEOPARD project considers each of the above issues in turn and proposes and implements corresponding APIs.

2.5. Tools Layer

Independent of the implementation and run-time aspects, reliable applications running on parallel systems require in-depth analysis of the correctness of the implementation and of time-related aspects such as schedulability, lack of deadlocks, race conditions, etc. With the use of multicore systems, parallel execution becomes the norm such that errors that do not manifest as faults on single CPU systems are much more likely to cause system failure.

For the correctness analysis of parallel applications, existing analysis tools need to be enhanced to find errors related to parallel execution through static analysis or concurrent unit tests. Such tools will are being developed in the JEOPARD project.

2.5.1. Data-Flow Analysis tool Veriflux. The data-flow analysis tool Veriflux, that was developed within the HIJA project [7] will be extended for the needs for parallel systems. This includes the detection of possible deadlocks, race-conditions, etc.

For the static analysis of these error conditions, the data flow analysis needs to use information on synchronisation as part of the context information produced by the data flow analyser for each call. With the availability of this context information, it will be possible to detect if synchronisation is used properly.

The static analysis tool will have to be applied to real applications to analyse its performance and reduce the number of *'false positives'* by enhancing the tool with support for typical code patterns that are difficult to analyse.

2.5.2. FRESCOR contract model. Real-time embedded software is becoming more and more dynamic and flexible. In order to cope with such developments, component-based real-time embedded systems have been studied. In such systems, components are developed, analysed and tested independent of each other. When they are put together, analysis and testing can be made without knowing any internal information of any component because the impacts of one component on the others are independent of its internal behaviour. Therefore, it is possible to make schedulable entities with very different timing requirements or criticalities coexist without jeopardising each other's execution. Utilising execution time servers to implement component-based real-time embedded systems is a common practice on uniprocessors.

Based on such techniques, Harbour proposed *FRESCOR service contracts* in a FRESCOR (Framework for Real-time Embedded Systems based on COntRacts) project report [6]. In a FRESCOR system, every application or application component should have a contract with the rest of the system, specifying the minimum resource requirement of this application or application component. Negotiations are made across the whole system, either offline or online, to guarantee, if successfully negotiated, that enough resources are always available to satisfy the minimum requirements of all contracts.

The FRESCOR contracts are devised to be platform independent such that any scheduling method can be used to support such contracts. Currently, FRESCOR contracts are mapped to execution time servers at runtime and all the tasks within an application or application component are executed under the server that represents the corresponding service contract. This means that the FRESCOR contract model is a hierarchical scheduling framework, which includes the scheduling of execution time servers and the scheduling of tasks within those servers.

One goal of the JEOPARD project is to extend RTSJ (Real-Time Specification for Java) to support the FRESCOR contract model at two levels. First, each application should have a contract with the underlying operating system which specifies a portion of resources to be reserved (i.e. external contracts). Second, components of each application should have their own contracts as well (i.e. internal contracts). Such contracts specify how to distribute the resources reserved according to the corresponding external contract among all the components.

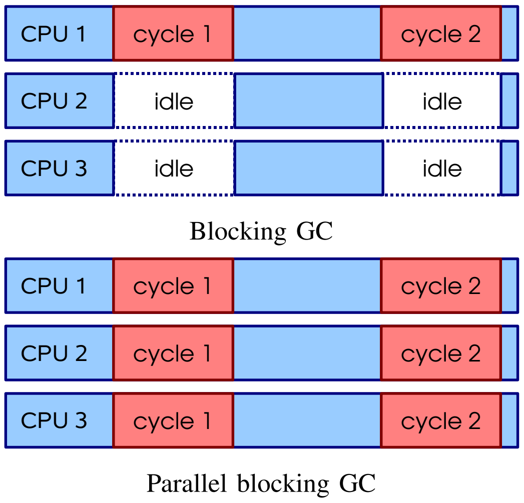

Figure 2. *A blocking garbage collector cannot be used in real-time systems, independent of whether it is a single processor or parallel garbage collector.*

3. Parallel Real-Time GC

The biggest challenge for a parallel real-time a Java VM is to provide real-time garbage collection that can execute in parallel and without preemption of the Java application. A blocking GC cannot be used in a real-time system due to the pauses introduced by GC activity, see Figure 2. For the use in a real-time system, it is of little importance whether a single-threaded or a parallel GC is used, both versions of a blocking GC will stop the application from being able to give timing guarantees.

3.1. Categorising Parallel GCs

In a real-time SMP Java implementation, at least, a parallel concurrent GC that uses a set of CPUs reserved for garbage collection work is required. Then, the system designer could assign a subset of the available CPUs to for the garbage collector. However, this bears two problems: the number of CPUs assigned to the GC must match the worst-case allocation behaviour of the application, which is difficult to determine statically. Assigning too little CPU power to the GC would mean that it might not catch up with the application, while assigning too much will waste resources. Therefore, we favour a more flexible approach by developing an incremental parallel GC as shown in Figure 3. Such a GC can be executed work-based depending on the application's allocation behaviour without wasting system resources when allocation load is low.

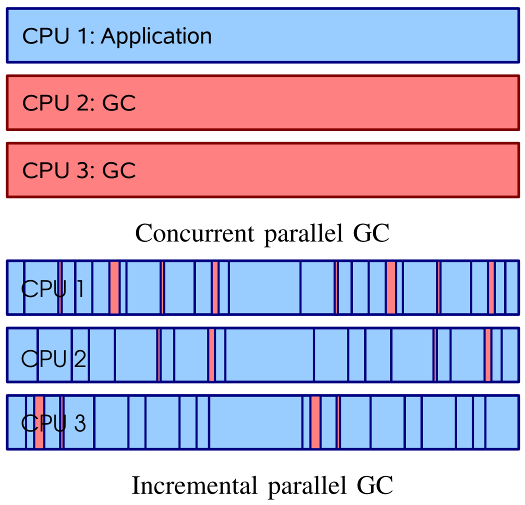

Figure 3. *A concurrent parallel garbage collector can use use a subset of the available processors and run in parallel of the application. However, most flexible is a parallel GC that can be fully interleaved with the application execution, i.e, a fine-grain incremental parallel GC.*

Such a real-time garbage collector must ensure that sufficient free memory is available even though many processors may be performing memory allocations and garbage collection in parallel. Fine grain synchronisation techniques need to be developed to ensure that this parallel execution does not suffer from contention. In addition to the implementation of a parallel real-time garbage collector, a theoretical analysis of the level of parallelism that can be achieved is needed. First results show that the theoretically achievable level of parallel GC depends on the heap layout formed by objects allocated by the application [14].

3.2. JEOPARD Real-Time GC

The basis for the incremental parallel real-time GC is the single CPU real-time GC used by the JamaicaVM real-time Java implementation [13]. This garbage collector uses an object model based on fixed-size blocks to avoid the need to compact the heap. It uses a Dijkstra-style three-colour mark and sweep algorithm [5], with the mark and sweep steps operating on single fixed-size blocks such that the time for a single step is bounded. To ensure that sufficient GC work is done, a work-based approach is taken, i.e., GC work is performed at allocation time only. For any application that uses a given fraction of the heap as reachable memory, a worst case for the GC work needed to allocate one fixed-size block can be given. A write barrier that marks any newly created references is used to ensure that heap modifications made during a GC cycle won't result in the reclamation of reachable memory.

The following paragraphs explain the aspects that need to be modified for it to become a parallel real-time GC.

3.2.1. Write barrier. The write barrier for the parallel collector must be a so called snapshot-at-beginning write barrier. I.e., it will mark all references that are overwritten by a new value, such that any object that is reachable at the beginning of the GC cycle will be retained by the GC. Marking a block means first checking, if it is unmarked (*white*) and then adding it to the set of *grey* blocks. The change to this write-barrier is required since the original write-barrier that only marked newly created references would not be sufficient when several threads would modify the heap in parallel.

3.2.2. Mark. The mark phase needs to permit several CPUs to perform GC mark steps in parallel. A mark step means taking one element *b* form the *grey* set, marking all unmarked objects referenced from *b grey* and marking *b* itself black. If the data structures on the heap cannot be scanned in parallel (e.g., the heap contains a long singly linked list), the parallel mark phase might not make progress on an arbitrary number of processors since the grey set may not contain enough elements to be marked in parallel. Luckily, such stalls during the mark phase will occur infrequently on typical applications [14].

3.2.3. Grey set. The grey set is accessed in parallel by several CPUs that may remove elements from this set or add elements via executing write barrier code or performing mark steps. A global list with synchronisation using locks or similar will not scale well. Therefore, CPU local grey lists are used and a compare-and-swap (CAS) based mechanism for load balancing in case a CPU local list is empty.

3.2.4. Finalisation. The finalisation phase needs to take care of Java objects that have a *finalize* routine or that are reachable via weak-, soft- or phantom-references. This is the shortest phase in the GC cycle. A global list of finalisable objects is used here, and CAS operations are used to access this list. This works well on small numbers of processors, for massively parallel execution, CPU local structures will have to be used to avoid frequent CAS retries.

3.2.5. Sweep. The sweep phase needs to ensure that the memory of all objects that remained unmarked (*white*) will be reclaimed and added to the free lists. For this, the whole heap needs to be scanned, which can be done in parallel fairly easily by assigning different heap ranges to different CPUs. All CPUs that perform sweep in parallel will access the global free list in parallel.

3.2.6. Free list. The free list is not only accessed by CPUs performing sweep steps that add blocks to this list, but also by all CPUs that perform object allocation and need to remove blocks from this list. Therefore, a global list with synchronisation will cause too much contention. Instead, it was chosen to use CPU local free lists of a limited size. Only when the CPU local lists are empty or full, their contents are refilled from a global list or spilled to this global lists. This exchange with the global list can be done using CAS such that no locking is required. Since refilling and spilling happens infrequently, the likelihood of a retry after a failed compare-and-swap is unlikely and the worst-case number of retries is limited by the number of CPUs minus one.

3.2.7. GC Phases. GC phase changes need to be performed whenever a phase (mark, finalize, sweep) is finished. These phase changes are performed using a compare-and-swap. In addition, for some phase changes, it must be ensured that all other CPUs that also perform GC work have finished their last step before any CPU can start doing a GC step for the new phase. For this, a simple notification scheme is used: the CPU that performs the mode change signals to the other CPU that it needs notification when its current GC step has been completed. In most cases, waiting for this notification can be done by busy waiting a short moment, only in case of preemption of the other CPU by the OS a more complex procedure is required.

3.2.8. Controlling GC work. With all the above issues solved, short worst-case execution times can be determined for single mark, finalize, or sweep steps. When to perform this work needs to be decided next. It has to be analysed now how such a collector can be used for the best profit of different kinds of applications on systems with different numbers of parallel CPUs. The possibilities are a purely dynamic work-based scheme performing GC work on allocation, or a statically assigning a subset of the available CPUs to perform GC work only. First experiments show that assigning some CPUs to perform GC work may significantly improve the average case performance, while the work-based approach makes analysis of the worst-case behaviour easier, since no knowledge about the allocation rates is required. A combination of both may provide good average case performance with easy to determine worst-case behaviour.

3.3. Parallel GC Results

All theses aspects together result in a fully parallel garbage collector. Parallel execution is only limited by the use of compare-and-swap operations that might require a retry. The worst-case number of retries, however, can be limited on an *n* CPU system as long as the time spent in between two compare-and-swap loops is at least *n* times as large as the time required for one iteration. The use of CPU local data structures ensures that this implementation can scale well to massively parallel systems.

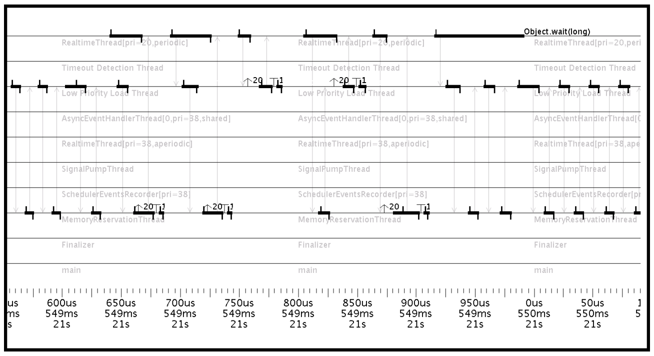

Figure 4. *An incremental GC that runs on a multicore system and uses locks to ensure atomicity results in frequent blocking of threads (thick lines are running threads, thin lines are blocked or ready).*

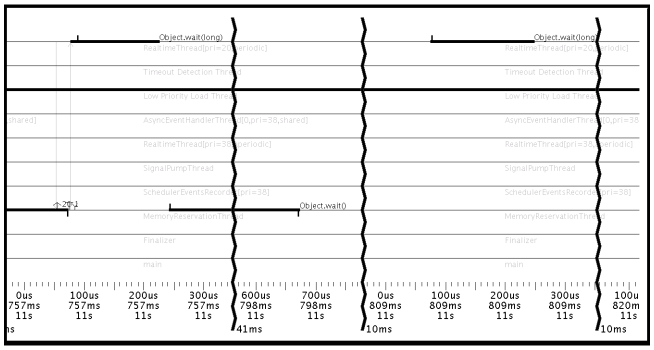

Figure 5. *A paralllel GC running the same application as in Figure 4 sees no preemption and no blocking of threads.*

When running a multi-threaded application on a multicore system, using non-parallel structures and locking results in severe slowdown due to threads blocking one another. This is shown in Figure 4: This graph illustrates the release of a high-priority task using an incremental GC whose incremental steps are made atomic via simple locking. Extremely frequent thread blocking and communication is the result, even on a two CPU system as in this example. A fully parallel GC that does not require locks to ensure atomicity of single GC steps does block any threads as shown in Figure 5: Here, the only form of synchronisation required is an occasional retry for a compare-and-swap. The analysis of when such retries will be required and what their worst-case behaviour is ongoing work.

4. Use Cases

Industrial partners within the JEOPARD consortium will apply the tools and technologies of the JEOPARD project for the development of real-world applications. The three industrial applications that will be developed are from the following domains:

1) **Multicore Radar Processor**: The tool chain will be used to develop an example radar processor based on a hybrid microprocessor-FPGA-system.
2) **Software Radio**: Multicore embedded processors enable the rapid reconfiguration of high-performance radio systems. The target is the validation of a system that can be rapidly reconfigured changing quickly the channel coding in order to adapt the device dynamically to the wireless communication systems features by selection among several advanced techniques such as Convolutional codes, Turbo-Codes, LDPC codes.
3) **Airline Operation Communication Solution**: On-board airplane component that handles communication related to flight plan and other on-board services. Applications have different safety levels and run in separate OS partitions.

5. Conclusion

The technologies developed by JEOPARD address the problems that arise when multicore systems will become widespread in embedded applications. First results show that an important performance gain can be expected from making use of several cores in parallel, even in systems that require real-time guarantees.

Modern multicore processors are optimised for best average-case performance, such that it is not surprising that theses systems show best performance benefits in non-real-time applications. However, even hard real-time systems can profit from these systems. The use of a parallel real-time Java VM provides flexibility that can be used to obtain better real-time behaviour compared to using only a single CPU: the multiple CPUs can be used to run services such as the garbage collector in parallel to the main application. If sufficient CPU time is assigned to the the GC, the application tasks will not see any pauses caused by the garbage collector.

Another result is that applications should be designed to make use of multicore systems. E.g., large linear data structure on the heap will make fully parallel garbage collection impossible resulting in limited scalability.

With the results of JEOPARD, multicore embedded real-time applications can be developed in Java. They could benefit from the same type of platform independence that is found in today's enterprise applications. These application will also benefit from a wealth of tools that is nearly unsurpassed. A majority of industries in the area have declared interest.

Many business critical applications (home applications, telematics applications, consumer applications) are already developed in Java. Actually most applications which involve service delivery are probably already based on Java because of the need to deliver the same applications to millions of different consumer systems (write once, run anywhere). Major frameworks and profiles have been defined for such applications: DVB-MHP for content based applications, OSGi for control based applications, MIDP for mobile devices. But as the services and applications to be deployed become more demanding on the computational power, it will be necessary to provide multicore real-time virtual machines that will enable the operation of these applications in a dependable manner. JEOPARD's contribution is a requirement to enable the mass deployment of such real-time embedded applications.

Acknowledgements

This work was partially funded by the European Commission's 7th framework program's JEOPARD project, number 216682. Contributions came from all project partners, in particular from Jose Almeida, Neil Audsley, Jacques Brygier, Annalisa Durantini, Scott Hansen, Thomas Mahr, Vlad Olaru, Ingo Prötel, Martin Schoeberl, Tobias Schoofs, Gabor Szeder, and Andy Wellings.

References

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18 2006.

[2] Greg Bollela. *Real-Time Specification for Java*. Addison-Wesley, 2001.

[3] Andrew Borg, Neil Audsley, and Andy Wellings. Real-time Java for Embedded Devices: The javamen project. In *Proceedings of the ERTSI 2005 - Embedded Real-Time Systems Implementation Workshop, held in conjunction with 26th IEEE International Real-Time Systems*, number YCS-2005-397 in Appears as a University of York Technical Report, pages 26–33, Miami, FL, USA, December 2005.

[4] Peter Dibble. JSR 282: RTSJ version 1.1.

[5] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. New York, 1976.

[6] Michael González Harbour. Architecture and contract model for processors and networks. Technical Report DAC1, Universidad de Cantabria (Frescor Project), 2006. http://www.frescor.org/index.php?page=publications.

[7] HIJA, High-Integrity Java, Project Number IST-511718 of the sixth framework programme of the European Commission. www.hija.info, 2004-2006.

[8] Jamaicavm. www.aicas.com/jamaica, 1999-2009.

[9] R. Kaiser and S. Wagner. Evolution of the pikeos microkernel. Technical report, January 2007.

[10] Robert Kaiser. Alternatives for scheduling virtual machines in real-time embedded systems. In *IIES '08: Proceedings of the 1st workshop on Isolation and integration in embedded systems*, pages 5–10, New York, NY, USA, 2008. ACM.

[11] Christof Pitter and Martin Schoeberl. Towards a Java multiprocessor. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 144–151, Vienna, Austria, September 2007. ACM Press.

[12] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[13] Fridtjof Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. aicas Books, 2002.

[14] Fridtjof Siebert. Limits of parallel marking garbage collection. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 21–29, New York, NY, USA, 2008. ACM.

[15] A.J. Wellings. Multiprocessors and the real-time specification for java. In *Proceedings of the 11th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing ISORC-2008*, pages 255–261. Computer Society, IEEE, IEEE, May 2008.