RJOP - A Customized Java Processor for Reactive Embedded Systems

Muhammad Nadeem
Department of Electrical and Computer Engineering
University of Auckland
Auckland, New Zealand
mnad011@aucklanduni.ac.nz

Morteza Biglari-Abhari
Department of Electrical and Computer Engineering
University of Auckland
Auckland, New Zealand
m.abhari@auckland.ac.nz

Zoran Salcic
Department of Electrical and Computer Engineering
University of Auckland
Auckland, New Zealand
z.salcic@auckland.ac.nz

ABSTRACT

This paper presents a novel, high performance and low cost execution architecture for the system level GALS programming language SystemJ, which extends Java with synchronous reactive features present in Esterel and asynchronous constructs of CSP (Communicating Sequential Processes). The new architecture is based on JOP (Java Optimized Processor), which is a hardware implementation of the Java Virtual Machine (JVM). The JOP, inherently suited to data-driven transformational operations, is extended to efficiently execute the control constructs and control flow of SystemJ. The new core, which is called RJOP (Reactive JOP) supports efficient execution of both data dominated and control dominated embedded applications. It also maintains the time-predictable execution of the applications intended for real-time embedded systems and calculation of Worst Case Reaction Time (WCRT) as provided by the original core. The initial results indicate significant performance improvement and lower resource requirements over the existing architectures used for the SystemJ execution.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and Embedded Systems

General Terms

Design, Languages

Keywords

Reactive Embedded Systems, Java Processor, Synchronous Languages

1. INTRODUCTION

The rapid growth in semiconductor technology which provided larger die sizes due to shrinking feature sizes have enabled the implementation of larger systems with more computational units. New application areas of embedded systems such as networked computing are composed of several computing units thus making them distributed systems in nature. They also tend to interact with the environment making them reactive by repeatedly reading inputs, doing computations and generating outputs. These computational units have different response times, hence, they need to run concurrently at different speeds. Also, it is important for these systems to have exact timing information or an upper bound of the execution time.

Programming such systems typically requires the use of non-standard control flow constructs, concurrency and exception handling. Among the programming languages, C, C++ and Java are the most commonly used in embedded systems. They are excellent for data computation but lack an easy way to make a program react to stimuli. The use of these languages typically leads to non-deterministic program behavior with respect to time and functionality. Also, the frequent occurrence of context switching to support concurrency reduces their efficiency. The synchronous languages can support the control flow constructs and synchronous concurrency whereas asynchronous languages support concurrency as well as the asynchronous communication. But both types of languages have poor data handling.

SystemJ [2] is a system level programming language based on the Globally Asynchronous Locally Synchronous (GALS) model of computation and allows the asynchronous coupling of synchronous reactive modules at the top level, which execute at different speeds. It extends Java with Esterel-like [1] constructs for the synchronous concurrency and reactivity, and CSP-like [3] constructs for the asynchronous concurrency. SystemJ targets a large range of heterogeneous embedded systems that combine data-intensive and control-dominated computations in addition to synchronous and asynchronous concurrency.

SystemJ programs can be compiled and executed using various approaches. The earliest approach translates the SystemJ program to Java source using TReK [2], which is a run-time Java library. But support for some reactive constructs was missing in this approach and dependence on Java multi-threading introduced undesirable non-determinism. In another approach the SystemJ program is transformed to an intermediate format called AGRC (Asynchronous GRaph Code) and the compiler back-end produces a single threaded Java code [5]. But this results in a large generated code size and slow execution time. Specific processors such as

DAC'11, June 5-10, 2011, San Diego, California, USA

ReMIC [6] has been proposed to handle reactivity, but it requires a specific compiler for porting of JVM to execute SystemJ.

The approach adopted in [5] separates the control-driven and data-driven operations and they are executed on two virtual machines running in parallel which results in reduced code size. However, this approach cannot achieve higher gains in execution speed as the JVM and CVM use Java Native Interface as the mechanism for communication. The latest approach reported in [4] uses a custom processor for implementation of the control-driven part instead of using the control virtual machine. The custom processor, called Control Processor (CP) executes the control-driven operations while any general purpose processor can be used to execute the data-driven constructs described in Java thus it is called the Data Processor (DP). The system may use one or more CPs in conjunction with one or more DPs and a system with one CP and one DP is called Tandem Processor (TP). This is a multiprocessor system that executes SystemJ programs relatively slowly due to the adopted communication mechanism between the control and data processor. None of the execution platforms mentioned is capable of predicting the execution time for the SystemJ applications.

Using a Java processor results in higher performance and predictability of the execution time. This paper presents a new processor core having architectural features to support reactivity and control flow to accelerate the execution of SystemJ programs in time-predictable way and more efficiently. The new core, named RJOP (Reactive JOP) is an extension of the JOP [7] and is inherently suited to data-driven transformational operations of SystemJ that are translated to Java. The RJOP extends the JOP with new capabilities to efficiently execute the signal manipulation and control constructs required to implement the reactivity and control flow in the synchronous part of SystemJ. It also allows to calculate the WCRT of synchronous clock-domains, which further can be used in efficient scheduling of full multiclock-domain GALS SystemJ programs.

Our main contributions presented in this paper are:

- First, extending and modifying the JOP to support SystemJ reactive constructs by introducing new byte codes and micro-instructions.
- Modifying the SystemJ compiler back-end to produce compatible Java codes for the modified JOP and to support future customization of the core.
- Supporting time-predictable execution of SystemJ programs.
- Better performance, code density and resources usage compared to single processor (executing Java only) and tandem-processor approach for SystemJ execution, thus making it more suitable for reactive embedded applications.

The rest of the paper is organized as follows. Section 2 provides a brief introduction of SystemJ followed by the overview of the JOP in Section 3. The new architectural extensions for the RJOP and some experimental results are presented in Sections 4 and 5 respectively. Conclusions and future directions are given in Section 6.

2. SYSTEMJ OVERVIEW

For the completeness of the paper, we briefly review the SystemJ and refer the readers to [2, 5] for a detailed description of the language. A SystemJ program consists of clock-domains (CDs), which have asynchronous concurrent behaviors, coupled using the asynchronous parallel operator (><) and communicating using CSP style rendezvous on channels. Each clock-domain consists of one or more reactions, which have synchronous concurrent behaviors. They are combined using the synchronous parallel operator (||) and execute in lockstep with the logical clock. The communication between the reactions of different clock domains takes place through the point-to-point *channels* whereas communication among the reactions of the same clock domain is carried out using *signals*, which are broadcasted synchronously.

The clock domains in SystemJ program are considered to execute as a sequence of clock ticks. The processing of all events arriving within a single tick are assumed to happen instantaneously or in zero time, which is referred to as the perfect synchrony hypothesis. The details of synchronous constructs can be found in [1].

3. JOP - JAVA OPTIMIZED PROCESSOR

JOP is a hardware implementation of the JVM targeted for small real-time embedded systems. The JOP is implemented as a small soft core that fits in an FPGA. It executes Java byte-codes much faster without JIT-Compiler [8]. The JOP architecture consists of the processor core, a memory interface and a number of IO peripherals as shown in Figure 1.

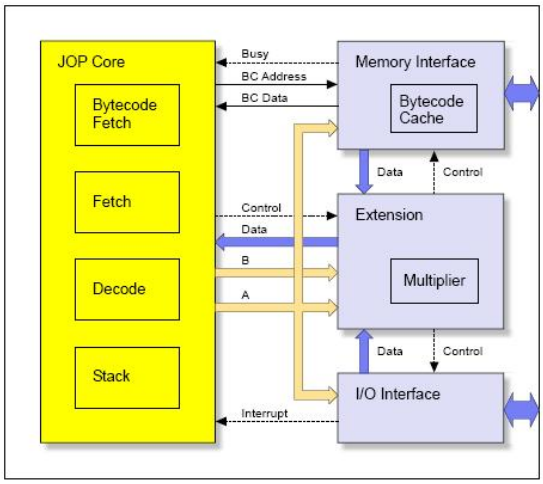

Figure 1: JOP Architecture: core, extension, memory and IO interface [8].

The processor core interacts with the memory through memory interface. The JOP has a stack cache as a substitution for the data cache and a method cache to cache the instructions. The method cache is organized in a way that it caches complete Java methods. A cache fill from main memory is only performed on a miss on method invocation or return. Therefore, all other byte-codes have a guaranteed cache hit. The memory interface provides a connec-

tion between the main memory and the processor core. The IO interface contains peripheral devices, such as the system time and timer interrupt, a serial interface and application-specific devices.

The JOP translates the byte-codes into sequence of its own instructions, called *micro-instructions*, which are 8-bits long with two extra bits indicating *opd* and *nxt* fields. These fields indicate the fetching of an operand and next byte-code respectively. The complete micro-instruction set of JOP along with the operations performed is given in [8]. Each Java byte-code is translated to a single or to a sequence of micro-instructions stored in a ROM. The mapping between the Java byte-code and the JOP micro-instructions is done using a jump-table generated during the application building. Each byte-code acts as an address for the jump-table and the corresponding location contains the start address of micro-instruction sequence for that byte-code.

The JOP core has four pipeline stages: Byte-code Fetch, Fetch, Decode and Execution (or Stack). In the Byte-code Fetch stage, the Java byte-codes are fetched from the memory. In the Fetch stage, micro-instructions are fetched, which are decoded and control signals are generated in the Decode stage. The address for the stack RAM is also generated in the same stage. In the Execution stage, the arithmetic/logical operation is performed. The operands are the top two elements of the stack which are stored in registers A and B and the result is stored back in register A.

4. RJOP - REACTIVE JOP

In all the approaches discussed in Section 1, the data computations are handled in Java. Hence, any performance improvement requires speeding up the execution of Java byte-codes. The JOP executes Java byte-codes faster when compared to a general purpose processor (upto 500 times) and other Java processors (upto 20 times) and uses less logic resources than these processors for its implementation. This makes it an attractive choice for the execution of SystemJ programs. In its existing implementations, it can be used to execute SystemJ programs directly:

- In pure Java single processor implementation of SystemJ, any standard processor can be replaced by the JOP.
- In a multiprocessor approach, it can replace the standard processor handling data-computation and work in conjunction with the control processor (CP) [4] to execute the SystemJ.

The single processor implementation is good for data dominated computations but it cannot handle the control flow constructs and signal manipulation well. It requires method calls when translated to Java thus incurring calling overhead. The multiprocessor approach is faster when compared to the single processor approach but it may need too much hardware resources. Also, the communication overhead can be a serious issue.

We are adopting an approach which provides a balance between performance, logic resources usage and power consumption as well as supporting real-time embedded applications due to its time-predictable nature. The current implementation of the JOP allows the extendability of the functionality by introducing custom byte-codes. We exploit this capability and extend the existing core to support reactive constructs, which help avoid communication overheads among the control and data processors as it is the case with the TP approach.

4.1 SystemJ Compiler Modifications

The existing SystemJ compiler, which produces Java code, needs modification for the code to be compliant with the new architecture. The SystemJ program is compiled to produce modified Java code with all reactive constructs being translated to special Java methods. These are not real methods and are substituted by special byte-codes on application building from application classes. The JOP is extended to accommodate new byte-codes and new micro-instructions are introduced along with the existing ones to support the the efficient execution of the reactive constructs of the SystemJ program. The resultant implementation provides the freedom to efficiently run reactive applications along-side the normal Java application code on a single processor.

The synchronous reactive constructs operate on signals which can have only status or both value and status. The placement of these signals in data structures is a critical issue. In the previous work, the signals are translated to objects of class *Signal* and the status of the signals is set by calling methods of this class, incurring calling overheads which results in slow execution. In the TP approach, the signals are implemented in data-memory of the control processor and any operation on the signals requires memory access which can incur multi-cycle latency depending on the hardware platform and memory used. Also, it does not perform ALU operations on the memory operands, so they need to be loaded into some temporary storage which increases the latency [4]. In addition, frequent access to large memory blocks results in more power consumption.

The reactive systems are characterized by the frequent interaction with the environment or, in case of GALS systems, interaction of reactions within the clock domain through signals. In the proposed approach, the signal statuses are stored in an array of processor registers, called *Signal-File* for faster and more energy- efficient access. The *Signal-File* is parametrized up-to 256 signals. When there is any signal dependency in SystemJ reactions, the execution of the current reaction is suspended and another reaction is scheduled. Once the signal dependency is resolved, the reaction is resumed from the same location where it was suspended. This is done by using the signal locks [4], which are also implemented in the *Signal-File* for faster access.

The modified compiler, still generating single threaded Java code, translates the pure SystemJ signals into simple Java variables instead of objects of a class and they are assigned a unique identity code. The signal emission and signal status-checking statements, which resulted in Java method calls in earlier version of the compiler, are translated into Java methods with the *Native* prefix recognizable by the JOP tools and are subsequently replaced by custom byte-codes.

All of the control flow constructs are translated into *if* and *else* statements which check the presence of the signal or the signal expression. The signal checks are carried out on these signal variables without invoking the methods of another class. Communication with the environment is provided through memory mapped IO. This includes loading input signals from the environment, emitting output signals and

Table 1: Compiler Translations from SystemJ to Low Level Java Statements

SystemJ	Java	RJOP Java
system { }	public class system { }	
signal S	Signal S = new Signal();	int S
emit S	S.setStatus( )	Native.emit(S);
present S	if (S.getStatus( )) term(p) else ;	Var = Native.present(S);
abort S	if (S.getStatus( )) term(p) else ;	Var = Native.present(S);
suspend S	if (S.getStatus( )) term(p) else ;	Var = Native.present(S);
reaction p(:) term(q)	public static void p()	public static void p()
synchronous concurrency	Switchcase...	Switchcase...
environment	S.sethook()	Native.lsip(id); Native.cer(); Native.lsop(id);
resetting	S.setClear()	Native.demit(S);
logical tick	tick();	Native.seot(); Native.ceot();

indicating end of tick etc. In the current implementation, acceleration of scheduling and asynchronous communication is omitted and is carried out in the existing way. The translation of SystemJ reactive constructs into Java statements using the original and the modified compiler are shown in Table 1.

Figure 2 shows the translation of SystemJ code to the standard Java source and the Java source compatible with the RJOP. A small example shown in Figure 2 (a) checks for the input signal A and if it is present then it emits signal B, otherwise C is emitted. Figure 2 (c) is the equivalent Java code where input signals are declared as objects of the *Signal* class. The checking and setting of signals is done by calling the methods of the class. Figure 2 (d) shows the modified Java code for the RJOP where signals are declared as integers and the setting and checking of the signals are performed by the related new byte-codes.

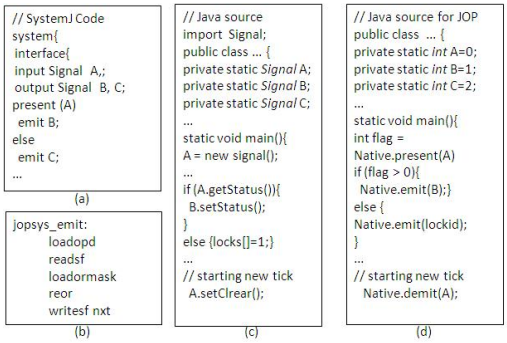

Figure 2: SystemJ source code translation to Java for JOP and mapping of *emit* byte-code to micro-instructions

4.2 Extending JOP

The JOP architecture is extended by introducing the new specific byte-codes to support the signal manipulation and control flow in SystemJ. Each byte-code op-code is one byte in length. There are 256 possible op-codes and not all of them are used. The instruction set of the JVM contains 201 different instructions [9]. Few of the op-codes have been used by the JOP to implement specific functions and the remaining can be used to implement the desired custom byte-codes.

Table 2: Compiler Translations from SystemJ to Low Level Java Statements

Byte-code	Task
jopsys_emit	Asserts the signal in a Signal-File
jopsys_demit	Reset signal in Signal-File
jopsys_initsf	Initializes the Signal-File contents if required
jopsys_present	Check the presence of the signal
jopsys_lsip	Load input signal from port to Signal-File
jopsys_lsop	Load output Signal from Signal-File to port
jopsys_seot	Marks the end of logical instant
jopsys_ceot	Clears the end of logical instant register
jopsys_cer	Clear environment ready signal

Table 3: List of the extended micro-instructions

Miscro-instruction	Task
readsf	Loads the contents of SF into register T1
loadormask	Loads the "or" mask into T2
reor	ORing of operands from T1 & T2
writesf	Write the contents of T1 into Signal-File
loadandmask	Load the "and" mask into T2
rdsip	Load input signal from IO to T1
reand	ANDing of operands from T1 & T2
cleartick	Clears the end of logical instant register
loadopd	Loads contents from A into T1
settick	Store tick value in T1
clrready	Store ready value in T1
pushonstack	Load from T1 to top of stack
popfromstack	Store top of stack in T1

Table 2 lists the new byte-codes introduced in the RJOP.

The existing set of micro-instructions is also extended with new micro-instructions to support these byte-codes as shown in Table 3. These micro-instructions are cost-effective, because they share the resources of the existing processor. They are used for data transfers among the existing and extended hardware components and perform arithmetic and logic operations. JOP accepts the interrupts at the byte-code boundaries and handling of interrupts in the core pipeline is avoided. The micro-instructions which construct one byte-code instruction are executed as an atomic operation.

Figure 2 (b) shows the mapping of a new byte-code to the micro-instructions. The method *Native.emit* is replaced by the *jopsys_emit* byte-code which is mapped to a sequence of six micro-instructions and the last one containing the *nxt*

field indicates the fetching of a new byte-code. This byte-code is used to set the status of a signal.

The JOP is extended with a custom hardware unit, called *Reactive Unit* to support new micro-instructions as shown in Figure 3. The decoder logic is modified to generate appropriate control signals for new micro-instructions and the ALU in the Stack unit is used to perform arithmetic and logic operations. This extension allows the RJOP to efficiently deal with the reactive applications along with the data-dominated computationally intensive applications. The byte-code fetch and micro-instruction branch units are modified to provide branching for the new unit.

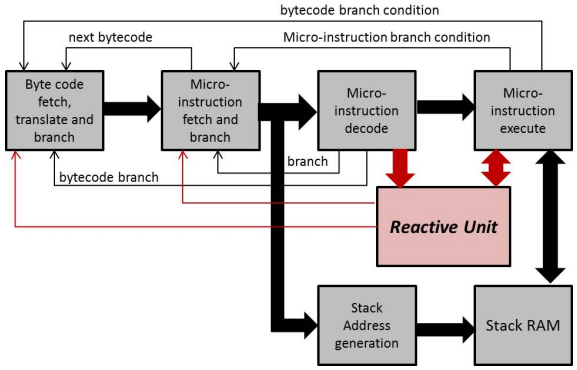

Figure 3: RJOP block diagram: JOP extended by Reactive Unit

4.3 Reactive Unit

Figure 4 shows the internal architecture of the Reactive Unit. It is composed of the *Signal-File* to store signal statuses, temporary registers to hold intermediate values (also act as an alternative for stack registers A & B) and a mask generation unit to mask or unmask a particular bit of a signal word. The masks are used for the setting or resetting of a signal. The Reactive Unit is interfaced with decoder, stack and internal stack cache. All of the signal handling byte-codes modify the status of the signals. The signal id is passed as a parameter in the *Native* method and is stored in the stack from where it is read into register A. The lsb byte is loaded from A into T1 by the micro-instruction *loadopd* The lsb 5 bits give the byte address whereas msb 3-bits indicate the location of the signal inside the byte and are also used to generate the mask for that signal. The signal word and mask are loaded into temporary registers T1 and T2 by *readsf* and *loadormask* micro-instructions respectively. The contents of both registers are ORed by *reor* micro-instruction and the result is stored back into T1 from where it is written back to the *Signal-File* by the micro-instruction *writesf*. In the case of the control flow instructions, the signal address is provided, and the signal is loaded into T1. The *andmask* is loaded into T2 which sets all other bits of T1 to zero. T1 is then checked for the presence of the signal. If T1 is not zero, the signal is present, else it is absent. The communication with the environment is carried out using both the existing and new micro-instructions. The data to be sent is pushed onto the stack using *pushonstack* from where it is stored in the data port register. The address is passed as a parameter and loaded into the port address register.

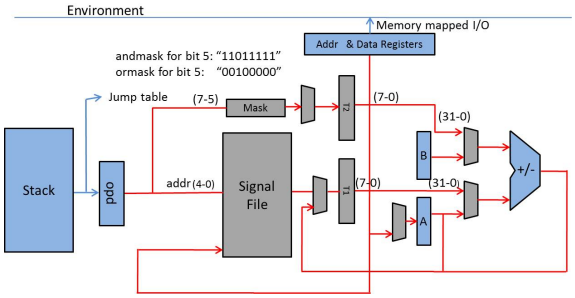

Figure 4: Internal architecture of the Reactive Unit

5. EXPERIMENTAL RESULTS

We have chosen four benchmarks from the same benchmark set used in [4] so that the results could be compared against the previous related works. The benchmark set consists of pure synchronous programs as they represent a special case of SystemJ program with a single program and are suitable for testing the synchronous reactive features of the architecture. The synchronous examples chosen include combinational lock (cl), demo loop (dl), runner and MP3 user interface model (Mp3ui) and are borrowed from Esterel test bench suite. The examples are mostly control-driven and have minimal data computation. All presented data have been collected from the experiments carried out by using the cycle-accurate ModelSim simulator and executing them on Altera Cyclone II FPGA with 70k logic elements, 2MB of RAM and running at 50 MHz clock. The system is capable of running at 100 MHz but the results presented are for 50 MHz clock for fair comparison with earlier published results. The performance of RJOP is compared against a standard general purpose processor (NIOS II), the TP [4] system and JOP [8]. Table 4 indicates the results for the general purpose processor and RJOP. The huge gains of the RJOP system compared to the GPP implementation (NIOS II in this case) can be attributed to the hardware implementation of the JVM.

Figure 5 shows the execution time for different systems in terms of the average tick time for each synchronous program, respectively. The RJOP is almost 2.7 times faster than the TP system. This gain in speed is due to two reasons. First, it is due to data-calls by CP to DP in a TP system as the TP system is loosely coupled and incur large communication overhead. Second, all the data computations are performed by the data-processor (NIOS II in this case) which is a general-purpose processor and has a software interpretation of byte-codes which is much slower than the hardware implementation. Hence, RJOP will always outperform the TP system when running applications involving data-dominated computations as the extensive data calls will slow down the system. In the case of TP, the scheduling of the reactions takes place in CP which is more efficient than the scheduling in single threaded Java code. It is expected that the introduction of the efficient scheduler will not only boost the performance but will also reduce the code size. The total logic element (LE) usage for all the execution platforms, when implemented on an Altera Cyclone II FPGA are shown in Table 5. The results show that RJOP uses slightly more logic than the JOP and is much

Table 4: Exceution Time (ms / tick) for RJOP and the General-Purpose processor

	cl	*dl*	*runner*	*Mp3ui*
GPP	1000	13	500	620
RJOP	1.0	0.3	0.5	2.9

Table 5: Hardware resource usage for different implementations

	NIOS II	**TP**	**JOP**	**RJOP**
LE	5256	7305	3514	3675

more economical compared to the other implementations.

The proposed approach provides currently the best platform for execution of SystemJ programs in terms of speed and size. The results show that the JOP is a better execution platform for Java-based data dominated computationally intensive applications in embedded systems. But, the SystemJ programs involving extensive signal emission and signal check statements tend to execute slower. The RJOP incorporates the capability to handle both the control flow and signal manipulation. The results in Figure 5 show that RJOP on average is 20% faster than JOP. The RJOP will perform even better if the application becomes more control oriented. New RJOP instructions that support reactivity all have bounded and finite execution times. Similar to JOP, the program worst case execution times can be found for RJOP. However, as RJOP executes SystemJ programs and reactions in lock-step with logical tick, it allows calculating the worst case reaction times (WCRT), which are the maximum time between any two consecutive ticks in any SystemJ clock-domain, which at the same time represents the minimum allowed time between two consecutive events from the environment. This feature makes SystemJ suitable for programming even hard real-time systems; however this is out of the scope of the current paper. Hence, RJOP is guaranteed to provide better performance regardless of whether the application is control-dominated or data dominated, and supports our idea of providing the hardware support for the reactive constructs of SystemJ language to harness its potential performance.

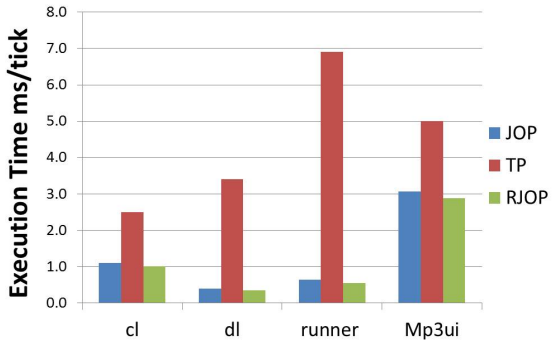

Figure 5: Execution times for synchronous benchmarks

6. CONCLUSIONS

We have described a new execution platform for programs written in the GALS programming language SystemJ. The processor core, an extension of the existing Java Optimized Processor (JOP), called RJOP, is enhanced with new instructions which support control-dominated operations of SystemJ and with native support for data-dominated operations. The new core as well as the required modifications to the SystemJ compiler and design flow result in significant improvements in performance over existing approaches for SystemJ execution. This is important because the original JOP has been successfully used in real-time embedded applications due to its ability to estimate worst case execution times, and the additional instructions of RJOP follow the same path offering predictable and much better support for reactivity and concurrency. As the next step we plan to extend RJOP with support of scheduling and asynchronous communication and use it as the basic building block for the multi-core system-on-chip for more efficient execution of the SystemJ.

7. REFERENCES

[1] G. Berry. The Esterel v5 language primer, April 1999.

[2] F. Gruian, P. Roop, Z. Salcic, and I. Radojevic. The SystemJ approach to system-level design. In *Proceedings of the fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, MEMOCODE '06*, pages 149–158, 2006.

[3] C. A. R. Hoare. *Communicating sequential processes.* Prentice-Hall, Inc., 1985.

[4] A. Malik, Z. Salcic, A. Girault, A. Walker, and S. C. Lee. A customizable multiprocessor for globally asynchronous locally synchronous execution. In *JTRES '09*, pages 120–129, New York, NY, USA, 2009.

[5] A. Malik, Z. Salcic, and P. S. Roop. SystemJ compilation using the tandem virtual machine approach. *ACM Trans. Des. Autom. Electron. Syst.*, 14(3):1–37, 2009.

[6] Z. Salcic, H. Dong, P. Roop, and M. Biglari-Abhari. REMIC - design of a reactive embedded microprocessor core. In *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, volume 2, pages 977–981 Vol. 2, 2005.

[7] M. Schoeberl. A java processor architecture for embedded real-time systems. *Elsevier Journal of Systems Architecture*, 42(1-2):265–286, 2008.

[8] M. Schoeberl. *JOP Reference Handbook: Building Embedded Systems with a Java Processor.* 2009.

[9] L. Tim and Y. Frank. *Java Virtual Machine Specification.* Addison-Wesley Longman Pub., 1999.