2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing

GALS-JOP – A Java Embedded Processor for GALS Reactive Programs

Muhammad Nadeem, Morteza Biglari-Abhari and Zoran Salcic
Department of Electrical and Computer Engineering,
University of Auckland
Auckland, New Zealand
mnad011@aucklanduni.ac.nz, m.abhari@auckland.ac.nz, z.salcic@auckland.ac.nz

***Abstract*—This paper presents GALS-JOP processor for efficient execution of programs written in SystemJ GALS programming language, which extends Java with both synchronous and asynchronous concurrency and directly supports the design of concurrent and reactive programs that comply with globally asynchronous locally synchronous (GALS) formal model of computation (MoC). In the first step, a Java optimized processor (JOP) is enhanced with a control processor (CP), which deals with concurrency and reactivity, to design an intermediate solution, called tandem processor, or TP-JOP, in which control processor and JOP work together to implement control flow and data operations of GALS programs, respectively. Then, JOP and the CP functionalities are merged into a single processor, GALS-JOP, which enriches JOP with some key constructs and abstractions for efficient implementation of SystemJ GALS programs. Experimental results demonstrate superiority of the new processor over all other approaches for implementation of SystemJ programs so far making it suitable for embedded systems.**

Keywords-component; GALS Processor; Reactive Processor; reactivity;concurrency; embedded systems

I. INTRODUCTION

One of the key goals in embedded systems design is efficient exploitation of concurrency as the way of reducing design complexity of the system and at the same time ability to deal with the requirements of timely response to the event coming from external environment. Significant research efforts have been made in tailoring Java and its execution environment to facilitate its use in embedded systems [1, 2]. However, dealing with concurrency is still left to the inefficient Java thread model. Java threads, besides low efficiency, are relatively unsafe model that require programmers to deal with low-level details of thread synchronization and communication resulting in programs with very little guarantees on worst case execution time. Also, Java memory model makes it difficult to implement it on simple processors. However, portability through the use of JVM and its variations are attractive features worthwhile of further exploration on how Java can be used efficiently in embedded world. One of the most notable approaches that makes Java relatively efficient is based on the use of dedicated Java processors, e.g. JOP [3], which besides efficient implementation of Java byte-codes in micro-coded sequences of elementary register transfers, offers very attractive feature of evaluating and guaranteeing worst case execution times for Java programs. However, JOP is primarily suitable for sequential Java programs and does not outreach to deal with Java concurrency in any special way. At the same time JOP, as an open source processor design, offers high level of modification flexibility in terms of instruction set architecture, memory model and changes in its data-path.

Recently proposed system-level programming language SystemJ [4] extends Java with both synchronous and asynchronous concurrency and directly supports the design of concurrent and reactive programs that comply with globally asynchronous locally synchronous (GALS) formal model of computation (MoC). Thus, SystemJ completely excludes Java concurrency and replaces it with its own. Computation within concurrent processes, which are mutually asynchronous on the top system level, and synchronous if within the top level processes, is performed using Java extended with a number of statements suitable for description of reactivity and pre-emptions. In its original implementation, SystemJ was compiled to Java and then executed on any processor that has a JVM port, or some variation of it, such as J2ME. However, SystemJ's powerful concurrency model in that case is implemented in Java and inherits some deficiencies, particularly those related to the lack of program flow control in the form of efficient *goto* mechanism.

An approach to avoid the shortcomings of Java when implementing concurrency and reactivity of SystemJ is to separate SystemJ program control flow, which includes concurrency, from the ordinary Java computations and then implement them on two separate processors, such as tandem virtual machine [5] or tandem processor [6]. All these implementations rely on the use of JVM on a standard processor to execute Java computations, and a specialized processor for execution of control flow. An attempt to enhance the efficiency of execution of SystemJ programs was to use a native Java processor, JOP [3], and make modifications to its instruction set to support reactive statements of the language. The resulting processor [7], increases the performance of SystemJ execution compared to JOP indicating further possible improvements, for the use of both JOP itself and SystemJ in embedded systems. However, that processor, by being just an extension of JOP, does not address directly concurrency as defined in SystemJ.

In this paper we present two approaches that result in a significant breakthrough towards the use of SystemJ for embedded applications. First, an intermediate goal of integration of JOP and Control Processor (CP) as a new

978-0-7695-4612-4/11 $26.00 © 2011 IEEE
DOI 10.1109/DASC.2011.67

tandem processor into a Tandem Processor based on JOP, or TP-JOP, is proposed. The approach exactly follows existing idea of combining Java Virtual Machine (JVM) [5] and Control Virtual machine (CVM) into a Tandem Virtual Machine (TVM), but eliminates the need of using JVM because JOP itself is a Java processor. Then, we analyze the TP-JOP and carefully merge a minimal set of features of the CP into JOP by extending JOP's instruction set, memory model and data-path. A new processor, called GALS-JOP, facilitates efficient execution of synchronous and asynchronous concurrency and reactivity (control flow) and Java oriented data computations by merging best of two worlds at low cost. Importantly, the design approach does not require any essential modification of the compilation flow of SystemJ [4], which is based on a formal semantics, giving advantages over non-formal programming languages and their compilation approaches.

Based on the described approach we at the same time provide a full design flow for embedded systems that use SystemJ, which has a number of notable features. First, it is aimed at the systems which are specified and will be implemented as concurrent programs that run on a customizable processor. Design specifications (programs) comply with the GALS MoC and as such can be formally analyzed. These programs, due to further checks during compilation, naturally lead towards software systems correct by the design. Finally, the way how the target execution platforms, GALS-JOP and TP-JOP as an intermediate solution, are designed guarantees finding the worst case execution times for any program segment, and in particular case of SystemJ programs worst case reaction times (WRCT), which are the worst times between any two consecutive logic ticks of any asynchronous part, clock domain, of a GALS program. These new SystemJ execution platforms are the major contributions of the paper, whereas a number of smaller contributions related to the design of the new processor are described in related sections. It should be noted that all processors used in comparisons are prototyped using RTL VHDL, synthesized and experimentally verified in FPGA implementation.

The rest of the paper is organized as follows. Section II introduces the global aspects of the proposed approach, which positions the presented work and its contributions. The evolution led to the new processor, GALS-JOP, is described and qualitative comparisons with related processors and execution models are described. The intermediate solution in the form of Tandem Processor based on JOP, TP-JOP, is also briefly introduced. GALS-JOP, which makes very fine merger of tandem processor approach into a single and more economical processor, is described in Section III. The details of modifications of JOP architecture, instructions set architecture of GALS-JOP and some implementation details are described in this section. Section IV gives qualitative comparisons of the series of processors that execute SystemJ and use JOP in some form, to indicate advantages of the GALS-JOP. Finally, Section V presents the conclusions and likely future works related to the extension of the GALS-JOP approach.

II. SYSTEMJ EXECUTION

This section provides background and related work analysis that lead to the GALS-JOP design.

A. Maintaining the Integrity of the Specification

A SystemJ program consists of multiple asynchronous processes, called clock domains (CD), which are described on the top design level. Each CD consists of a number of synchronous concurrent processes, which execute in lock-step, driven by a logical clock, called tick. These synchronous processes are called reactions. Behavior of the reactions fully complies with synchronous reactive (SR) MoC [8, 4]. Reactions communicate within a CD, as well as with the external environment (input/output) through signals, which are broadcasted and presented within the current tick. Communication between reactions in different clock domains, which are asynchronous each to the other, is carried out through the exchange of messages over channels, which are semantically the same as channels used in CSP MoC [9, 4]. Besides operations on signals and channels, SystemJ allows free use of Java data objects and statements in its reactions, and those statements are considered instantaneous in terms of logical time (i.e. they do not consume logical time or ticks). Control flow of a SystemJ program incorporates scheduling of all reactions and clock domains, as well as communication between reactions, and communication with the external environment. Instantaneous (Java) computations are typically called data computations and consist of continuous blocks of Java statements (called Java action nodes or blocks). Thus, SystemJ program execution can be considered through its control flow in which data computations are invoked by control code where necessary.

An example of a simple SystemJ program that consists of two clock domains CD1 and CD2 is presented in Figure 1. The program receives an input from the environment through signal S. Once signal S is present, reaction R11 emits an internal signal to another synchronous reaction R12 to send a prepared message to another clock domain CD2 over channel CH. The program code illustrates the use of both synchronous and asynchronous concurrency. CD2 contains single reaction R21, which uses a simple Java print statement to print the content of received message on system output.
SystemJ compiler [4] is based on an intermediate representation of the SystemJ program called Asynchronous Graph Code (AGRC). The front-end of the compiler produces AGRC graph of the program, which is then used by the back-end of the compiler to create the object code for the target platform. Figure 2 illustrates compilation and execution strategies of interest for embedded targets that take advantage of the separation of control and data computations and resulting compact representation, particularly of the control code. Tandem Virtual Machine (TVM) [5] has significant reduction of memory footprint and improves the execution speed of SystemJ programs compared to JVM only approach, but it still requires JVM for execution of Java data computations. A special Control Virtual Machine (CVM) executes the control part, which is using a dedicated instruction set architecture, by interpreting instructions of the

control part. The CVM and JVM co-operate while executing SystemJ program. This approach has been extended to a full dedicated Control Processor (CP) instead of CVM, enabling much faster execution of the control part, but at the cost of additional logic resources used to implement the CP. The resulting two-processor system that comprises of a general purpose processor, which executes the JVM, and the CP is called Tandem Processor, or TP [5].

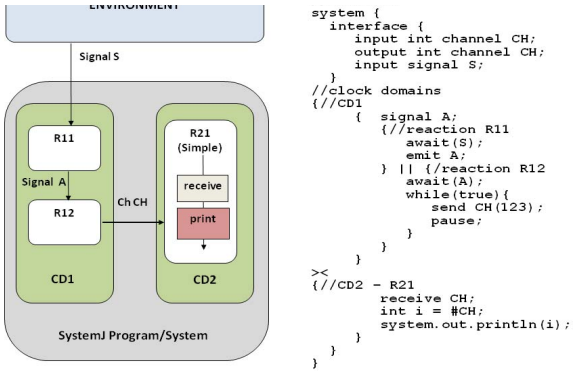

Figure 1. Example of a SystemJ program.

This paper goes beyond the original TP by replacing a general purpose processor with a Java processor, JOP [10], resulting in JOP-based tandem processor, TP-JOP. This approach is interesting in terms of its implementation and performance gains compared with the previously designed TP on one hand, and the GALS-JOP, which uses TP-JOP as its foundation, on the other hand. GALS-JOP merges control and data oriented processing into a single processor, which is based on JOP, but it is also enhanced with numerous features from the control processor (CP). GALS-JOP significantly increases cost-effectiveness of the resulting solution compared to all previous approaches and achieves the goal of more efficient execution of SystemJ programs on a small embedded platform.

B. *New SystemJ Execution Strategies*

The main goal of this work is focused on the SystemJ execution strategies and platforms that are suitable for embedded applications, which also indicate their potential of being used in real-time systems. The options presented in Figure 2 at the same time illustrate the evolution of the strategy that would optimize trade-offs of implementation resource complexity, speed and memory requirements. In that context GALS-JOP seems remarkably good solution. This work fully relies on the existing AGRC Compiler [4] and thus emphasizes portability of the compilation technology. The full compilation/design flow, where only additions to the compiler back-end are needed, is presented in Section III. A quick qualitative comparison of the execution strategies considered in this paper is given in Table I.

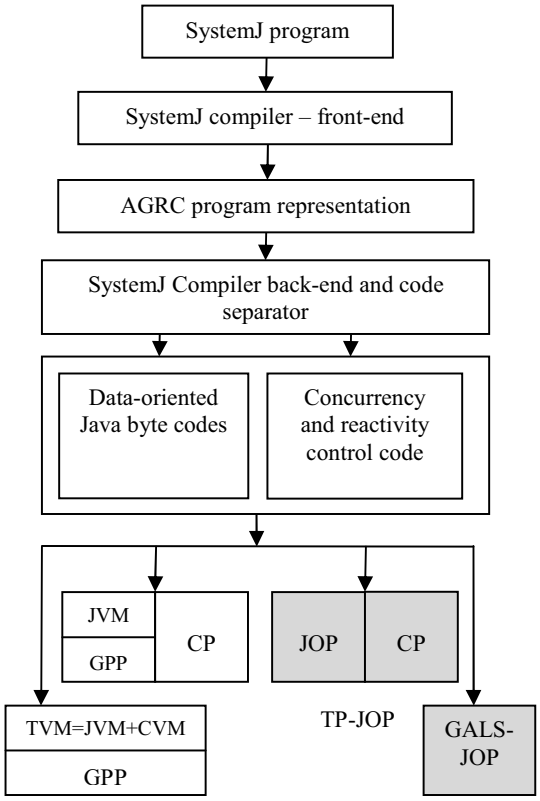

Figure 2. Compilation and execution target strategies.

C. *TP-JOP – a native execution approach*

TP-JOP is an intermediate solution that is implemented based on the idea of tandem execution of two processors, one that controls the flow of SystemJ program (CP, control processor) and one that executes operations that are within our GALS MoC (which are considered instantaneous and expressed in standard Java). The data oriented part of the code which is in Java was executed on a general purpose processor with JVM in previous implementation of the TP. The TP-JOP, as the name indicates uses full Java processor, and its basic model, which also indicates the execution of the control flow. At any time, each reaction can be executing either pure control statements or waiting on Java action block to be executed. However, if a reaction is waiting for

TABLE I. QUALITATIVE COMPARISON OF SYSTEMJ EXECUTION STRATEGIES

Qualities	Execution Strategies		
	JOP	***TP-JOP***	***GALS-JOP***
No. of Processors	1	2	1
No. of program memories	1	2	1(2)
Memory footprint	Large	Small	Small
Speed	Low	Fast	Fast-Med

TABLE II. CP INSTRUCTION SET

Instruction	*Register Transfers*	*Mode*
AND Rz Rx Operand	Rz <- Rx AND Operand	immediate
	Rz <- Rz AND Rx	indirect
OR Rz Rx Operand	Rz <- Rx OR Operand	immediate
	Rz <- Rz OR Rx	indirect
ADD Rz Rx Operand	Rz <- Rx + Operand	immediate
	Rz <- Rz + Rx	indirect
SUBVRz Rx Operand	Rz <- Rx - Operand	immediate
SUB Rz Operand	Rz - Operand	immediate
LDR Rz Rx	Rz <- Operand	immediate
	Rz <- M[Rx]	indirect
	Rz <- M[Operand]	direct
STR Rz Rx	M[Rz] <- Operand	immediate
	M[Rz] <- Rx	indirect
	M[Operand] <- Rx	direct
JMP Rx	PC <- Operand	immediate
	PC <- Rx	direct
PRESENT Rz Operand	if Rz(0)=1 then PC<-Operand else NEXT	immediate
SENDATA Rx	M[TP]<-Rx	indirect
CHKEND Rz Rx	Rz <-MAX{Rx[15:12], Rx[11:8],Rx[7:4], Rx[3:0], Rz[3:0] }	indirect
NOOP		inherent
SZ Operand	if Z=1 then PC <- Operand else NEXT	immediate
CLFZ	Z <- 0	inherent
CER	ER <- 0	inherent
CEOT	EOT <- 0	inherent
SEOT	EOT <- 1	inherent
LER Rz	Rz <- ER	indirect
SSVOP Rx	SVOP <- Rx	indirect
LSIP Rz	Rz <- SIP	indirect
SSOP Rx	SOP <- Rx	indirect
SWITCH Rz Rx	Rz <- M[Rx] Rz <- Rz + Rx + 1 PC <- M[Rz]	inherent

the result of Java computations, control point can be transferred to another reaction, thus actually not blocking the execution of control. The only point when control flow can be suspended is if all reactions have requested the execution of Java code and none has received the result of Java computation. This mechanism is implemented through a queue (FIFO) which stores the Java call requests and releases them in the order of their arrival. The effectiveness of the mechanism has been demonstrated on both virtual and physical implementation of tandem operation of control and data computation. TP-JOP has strict separation of control from data computations, which inevitably results in duplication of some of the computation resources, as shown in Table 1. The major cause for additional resources is the use of two processors. Although the CP is very simple but it duplicates basic instructions for data movement and arithmetic operations, as well as requiring two program memories in which control part and data part programs are stored. Also, major data structures that represent clock domains, reactions and objects of SystemJ, signals and channels, are contained in the data memory of CP. Only handfuls of instructions are specialized instructions that operate on these objects or are dedicated to support control operations related to AGRC-based control flow.

This was a major motivation to look more closely at how functions of the program control flow and interactions with the environment could be merged with JOP's processor functionalities resulting in a more economical and efficient implementation. The full CP instruction set, which includes AGRC-related operations presented in [5, 6] is given in Table II.

III. GALS-JOP

GALS-JOP can be viewed as a merger of JOP and CP in TP-JOP in addition to optimizations to implement SystemJ program execution on a single processor. JOP allows extensions of its instruction set, as well as the modifications of its data-path. Also, the decision to preserve SystemJ compiler in its entirety has been another motivation for this approach.

A. Principle of Design

The above explained strategy has been implemented by the following design principles:

Memory organization: GALS-JOP has a number of functionally-specific memories, which contribute to the faster SystemJ program execution. The main memory is allocated in two parts, program and data part. In the program memory part first a main method is stored, followed by the Java methods that execute Java data computations of individual clock domains corresponding to the action nodes of the AGRC graph. Control part of SystemJ program is mapped to the main method (also referred as control method) which is transferred to separate control (program) memory during the start up. This is illustrated in Figure 3. The remaining space in the main memory is used as a heap to store Java data objects and data structures [5] on which control methods operate, which is called the Control Array (CA). The CA is used to store data structures used to implement asynchronous (clock domains) and synchronous concurrency (reactions). Also, not shown in Figure 3, is the stack that serves as a temporary local storage for all data methods. Besides main memory, GALS-JOP has a number of additional memories for faster access to the program instructions. Method cache is inherited from JOP and it

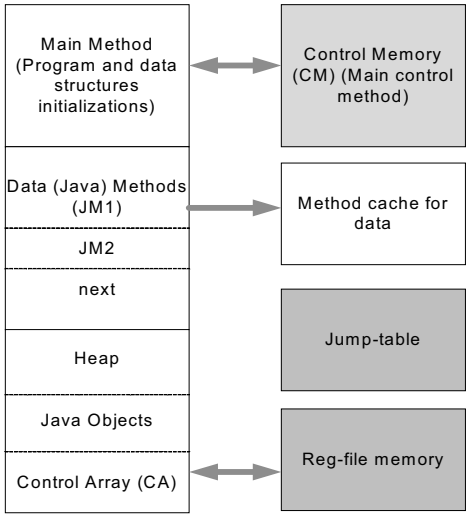

Figure 3. Compilation and execution strategies

always contains currently executing Java data method other than the main. On each invocation of another Java method, the method cache is loaded with the new method code to be executed. However, the SystemJ program flow is controlled from the control part of the program. In order to speed-up its execution, the control code is stored in separate control memory, which has the same characteristics as method cache. However, unlike method cache, once loaded with control part of SystemJ code, this memory is locked and does not change its content until the program execution is completed. It should be noted that all GALS-JOP memories, except the main memory, are implemented as FPGA internal memories and can be customized to the necessary application-specific size.

Instruction set: All instructions of the CP used to represent the program flow control are merged into JOP ISA, except those which already have identical execution semantics in JOP. This resulted in 21 additional byte-codes when compared to JOP. GALS-JOP has a new memory block called Jump- able, which contains the jump-addresses used in the control method. This helps executing complex control flow much more efficiently compared to the case with usual Java statements which do not support *goto*.

TABLE III. BYTE-CODE EXTENSION

Byte-code	***Description***	
jopsys_ldrind	Jopsys_strimm	Jopsys_aluimm
jopsys_ldrdir	Jopsys_jumpimm	Jopsys_aluind
jopsys_strsop	Jopsys-subf	Jopsys_cabaseaddress
jopsys_dummy	Jopsys_present	Jopsys_ldio
jopsys_jumpind	Jopsys_sz	Jopsys_strsop
jopsys_strdir	Jopsys_chkend	
Jopsys_strind	Jopsys-switchjump	

Instruction micro-codes: They implement register transfers in JOP's data-path needed by new bye-codes and are introduced in micro-code memory of JOP [10]. Careful analysis of these register transfers resulted in only 23 new micro-codes, which are used in implementation of new byte-codes as given in Table IV.

TABLE IV. MICRO-CODE EXTENSION

Micro-code	Micro-code	Micro-code
stdmbaseaddr	wrctrl	pushonstack
popfromstack	settick	clrzf
wrrf	reslevel	dmaddr
rdrf	inclevel	srzf
lockmain	declevel	ctrlinitfinished
ldmaincontaddr	stmaincontaddr	findmax
ldmaininvokcheck	ldmainreturncheck	loadjmpaddr
ldmaininvokedcheck	aluop	

Optimization of data-path resources: All unnecessary resources are removed, e.g. FIFO used in TP-JOP, to enable transfer of control to/from Java computations and mechanism for return of results from Java computation to the CP. These functions are directly implemented in GALS-JOP data memory. The register-file is replaced with a single port memory now.

SystemJ compiler: The original SystemJ compiler is preserved with the additions to the back-end that enable merger of the data oriented Java and the control code. These additions operate on the code produced by the SystemJ compiler [4].

B. Instruction Set and Executions

The additional instructions incorporated into existing JOP's instruction set functionally correspond to the instructions of the Control Processor (CP), which are now adopted, and modified where necessary, into GALS-JOP instruction set. However, as the CP has certain number of instructions with identical or similar functionality to the existing JOP's byte-codes, those instructions are not included into GALS-JOP, thus resulting in a reduced total number of instructions compared to TP-JOP.

C. Control Flow Implementation

When JOP is the target for SystemJ program, all the Java methods are loaded into method cache first for faster access. A similar approach applies to the GALS-JOP with the exception of the main method, comprising of control flow instructions and calls to data computation methods, which are not loaded in the cache. Instead it is permanently loaded into another fast memory called Control Memory (CM). During the start-up procedure, the method cache is loaded with the byte-codes of the start-up method. Once start-up is completed, the main method is invoked. Invocation of the

main method does not require access to the method cache and its byte-codes are fetched from the CM, which is another source of byte-code now together with the method cache. All the methods invoked by the main, are loaded into and executed from the method cache until the control returns to main. Whenever main invokes a Java method, the address of the returning (next) byte-code is stored in a register, called *main_continue_address* (MCA), which is loaded into JOP's Program Counter (JPC) to resume the execution of the main from the same location where it left the main. Similarly, returning to the main is different from the return to other methods as it does not require loading of the cache with main method, which saves the time otherwise consumed in loading the cache. Hence, we have different strategies for invoking main and other methods which requires keeping track of method call nesting by using a Level Tracker (LT) which is enabled as soon as CM is initialized. Any method invocation increments the Level Tracker, and return decrements it. If Tracker is enabled and a method is being invoked with level 0, it means the main is being invoked, and level 1 means a current executing method has been invoked by the main. Similarly, when returning, the level is first decremented, and then it is checked. The return with level 1 means returning to the main. The Level Tracker Decoder (LTD) generates different check signals such as *main_invoked_check*, *main_invokes_check* and *main_returned_check* which are used to carry out different invoke and return strategies.

D. GALS-JOP Data-path

The GALS-JOP data-path is presented in Figure 4. The JOP architecture consists of the processor core, a memory interface and a number of IO peripherals. 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 internal RAM. In the Fetch stage, micro-instructions are fetched from the memory. In the Decode stage, micro-instruction is decoded and control signals are generated. The address for the stack RAM is also generated in the same stage. In the Execution stage, arithmetic/logic operation is performed. The operands are top two elements of stack stored in registers A and B and the result is stored back in register A. The GALS-JOP design extends JOP's data-path in an upward compatible fashion.

The additional components introduced in the data-path are: a register-file memory, a Jump-Table, counter, various other registers and multiplexers shown as shaded in the Figure 4. Most of the registers are used to store address information, whereas register T1 and T2 are two general purpose temporary registers used for data manipulation. The memory for CA is allocated on the heap by simply declaring the arrays in Java. The byte-code *jopsys_cabaseaddr*, stores the base address of array in the control array base (CAB) register, to provide direct access to the CA. Otherwise, each CA access will require expensive read of object reference and CA base address. Loading an immediate value into a register is the most frequently occurring operation. This is implemented by *jopsys_ldrimm* instruction which requires only two cycles for execution. The address of a register in the register-file memory and the contents to be written are provided as TOS (top of stack) and TOS-1. The loading of data from the CA to register-file is done by using the *jopsys_ldrdir* and *jopsys_ldrind*. All the CA addresses are relative to the base address and thus are added to the base address to find the physical address. The data is stored in CA using *jopsys_strimm*, *jopsys_strind* and *jopsys_strdir*.

In case of jump instruction, the target addresses are available in the Jump-Table. The index is provided on the TOS as argument which is used to read the required target address into T1. The conditional jumps are implemented through the *jopsys_sz* and *jopsys_present* where target address is loaded to T1 and data to be checked is pushed on TOS. The program counter is loaded with target address from T1 if the condition is satisfied. The *jopsys_switchjump* is the most complex and expensive instruction. The pointers to all the cases are stored in the contiguous memory locations in ascending order following the switch node. The address of switch node is read from the register-file memory. The switch node value read from the memory contains the number of the case to be executed. This value is added to switch node address and physical address is calculated. The value read from the memory is loaded into the JPC to jump to the desired location. The GALS-JOP interacts with environment through memory-mapped IOs containing a set of registers.

E. Compilation Flow and Implementations

The GALS-JOP compilation flow requires a number of relatively small additions to the back-end of the original SystemJ compiler. Java code produced by the SystemJ compiler (**.java*) is merged with the control code (**.asm*), also produced by the SystemJ compiler, into a code that consists of the pure Java code and Java native methods. Java native methods are assigned new byte-codes, which extend the original JOP byte-codes with the operations necessary to implement control flow efficiently. An Assembly to Java Translator (AJT) has been developed to perform this translation. The *.java* file is compiled by the *Javac* compiler which reads class and interface definitions, written in Java programming language, and compiles them into byte-code class files. Another tool called *JOPizer* [10] based on open source BCEL, links a Java application and converts the class information to the format that JOP expects. The result is a **.jop* file generated by *JOPizer*, which also provides mapping between the native methods and the special byte-codes. The tool Jump Table Generator (JTG) reads the **.jop* file and generates the Jump-Table content, which contains all addresses used as the target in special (new) byte-codes. The JTG looks for the byte-code representing the source label and each time it comes across such a byte-code, its corresponding address and index are saved in the Jump-Table. The final table size depends on the number of the labels. Hence each jump byte-code is accompanied by an index of the label as a parameter which is then used to read the target address from the Jump-Table, which is implemented in an on-chip RAM and initialized at the start.

IV. EXPERIMENTAL RESULTS

A number of programs have been used as benchmark to compare different SystemJ implementation strategies. It includes both purely synchronous and GALS data models. The benchmark set includes the synchronous examples such as *combinational lock* (cl), *demoloop* (dl) and *runner* whereas *asynchronous protocol stack* (aps) represents the asynchronous case as given in [5]. The *dl* and *cl* examples mainly check the synchronous and reactive features of the architectures as they contain extensive signal check and signal emission statements whereas *runner* example also contains a considerable data-computation apart from the above mentioned features. The *aps* example contains data communication between the clock domains and is heavy on data computations. All examples have single clock domain except the *aps* which has two clock domains (cd0 & cd1). Table V gives lines of the original SystemJ codes and numbers of clock domains and reactions in each case.

TABLE V. LINES OF CODE

Programs	Characteristics		
	SYSTEMJ (LOC)	*No. of CDs*	*No. of reactions*
dl	57	1	4
runner	97	1	4
cl	104	1	4
aps	147	2	6

The generated code size comparison among different target platforms for the chosen benchmark set is given in the Table VI which shows that the GALS-JOP is the most economical platform due to the following reason. In case of TP, each instruction is a 16-bit word and most of the assembly instructions are compiled into two instruction words. In case of GALS-JOP, instructions are byte long and each assembly instruction is translated to one or more than one Java byte-codes (depending on the number and type of arguments passed) and may be compiled to more than one byte-code. But, most of instructions are translated to a single or two byte-codes resulting in reduced memory footprint. Also few of the instructions required in TP are not needed in GALS-JOP e.g.; checking whether the data-call launched has been served or not. The generated code sizes given in the Table VI does not include the size of the JVM because of being common to all the approaches. The generated code size for GALS-JOP is on average 7% smaller than TP-JOP,

TABLE VI. GENERATED CODE SIZE (KB)

Programs	Execution Platforms				
	GPP	*TP*	*JOP*	*TP-JOP*	*GALS-JOP*
dl	25	8	7	7	6
runner	45	15	13	14	12
cl	80	31	45	26	23
aps	45	35	57	28	28

42% smaller than JOP, 22% smaller than the TP (1:1), and 65% smaller than the GPP.

The GALS-JOP performance, given in Table VI, is very close to the TP-JOP performance as might be expected. The translation of the assembly code to custom Java byte-codes sometimes requires a sequence of instructions resulting in the execution times that vary between 1 and 20 cycles, which are worse than the 3 cycles taken by the non-pipelined custom CP to execute most of the instructions. The performance of GALS-JOP being very close to the TP-JOP can be attributed to a number of factors. First, it can be attributed to the absence of the communication interface thus no communication overhead. Second, some of the most frequently occurring instructions take fewer cycles than custom CP, thus making the execution of control faster. Third, in TP-JOP, data computations are wrapped in a case statement. It means whenever a data-call occurs, it needs to traverse the whole case statement before it finds the required target, thus consuming more time to identify the required action node. The program running on JOP always polls the CP, and upon finding the data-call, it invokes the method containing clock domain data computations wrapped in the case statement which in turn, invokes the Java method for the data computations. The JOP has method cache which holds only one method at a time. The GALS-JOP efficiently handles this situation of return from the main and avoids loading of the main method into cache as mentioned earlier. Also, the data computations are decomposed into small methods (each case is wrapped in a method) which require less time for loading into the cache as compared to a single method containing all the data computation. The GALS-JOP is 2 to 10 times faster than the JOP, 3 to 53 times faster than the TP and 13 to 6000 times faster than the GPP (Nios II). The GALS-JOP approach is preferred as it uses a single processor to achieve the same results as of TP (two processor approach) which is more costly. This single processor approach is faster by an order of magnitude when compared to other single processor counterparts such as GPP (Nios II) and JOP itself. As expected, it is typically slightly slower than TP-JOP, which achieves its performance by using significantly more silicon (logic elements when implemented in FPGA). It is expected that GALS-JOP will outperform all other architectures when executing the data intensive applications as is the case with the *runner*. The results are presented as the average tick time.

TABLE VII. AVERAGE TICK TIMES (MILLI-SECONDS)

Programs	Execution Platforms				
	GPP	*TP*	*JOP*	*TP-JOP*	*GALS-JOP*
dl	13	3.40	0.39	0.07	0.10
runner	520	6.90	0.64	0.15	0.13
cl	1000	2.50	1.66	0.14	0.16
aps (cd0)	0.80	0.30	0.14	0.05	0.06
aps (cd1)	3.15	0.40	0.25	0.13	0.16

Table VIII shows the resource utilization for different target platforms in terms of the used logic elements when implemented on an Altera Cyclone II FPGA. The GALS-JOP uses 42% and 31% less logic elements compared with TP and TP-JOP respectively. The reduced logic element usage can be attributed to firstly, the JOP itself, which is an economical processor. Secondly, unlike the TP approach, no additional interface is required as both data and control are implemented on the same processor.

TABLE VIII. RESOURCE USAGE

	TP	TP-JOP	JOP	GALS-JOP
LEs	7350	6228	3579	4306

V. CONCLUSION AND FUTURE WORK

In this paper we introduced two new execution platforms for SystemJ GALS programming language. The first one is tandem processor, TP-JOP, that uses two processors to execute control (CP) and data dominated Java (JOP) parts of SystemJ programs, respectively. This implementation served as the starting point to arrive at the ultimate goal of a single processor suitable for efficient and economical execution of SystemJ programs. The new processor, called GALS-JOP, merges the features of control processor (CP) into Java processor (JOP) and, so far, represents the most efficient execution platform for SystemJ. A modified programming and memory allocation model, together with the enhanced instruction set based on the original JOP processor, resulted in very efficient implementation. Also, this implementation preserves all features of the original JOP and all new features built into the GALS-JOP preserve ability to calculate execution times of resulting programs, making possible analysis of SystemJ programs in terms of the worst case reaction times (WCRTs), where the WCRT is the longest time of any individual tick within SystemJ program. Those times can be calculated for each clock domain separately and used in the analysis of real-time features of SystemJ programs, which is one of our future goals. GALS-JOP also preserves existing SystemJ compilation design/flow with minimal additions to the back-end of the compiler. Its current implementation is the most economical in terms of required resources (in FPGA), while TP-JOP shows highest average performance (average tick execution time over the range of programs), slightly ahead of GALS-JOP.

Future work also includes incorporation of all GALS-JOP features into the existing design and program analysis tools of JOP. Also, we plan to include hardware support for scheduling of clock domains and consider other scheduling techniques. Power consumption, as well as the energy efficiency of the processor will be analyzed and potentially improved. Finally, GALS-JOP will be considered as the basic building block for implementation of multi-core GALS processor on programmable chip.

REFERENCES

[1] The Real-Time specification for Java, http://www.rtsj.org/

[2] T. Henties, J. J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, J. Vitek: Java for Safety-Critical Applications, Electronic Notes in NTheoretical Computer Science, Elsevier, 2009

[3] M. Schoeberl, A Java Processor Architecture for Embedded Real-Time Systems, *Elsevier Journal of Systems Architecture*, vol. 54, issue: 1-2, 2008, pp. 265-286

[4] A. Malik, Z. Salcic, P. Roop, A. Girault: SystemJ: A GALS language for system level design, Elsevier *Comput. Lang. Syst. Struct.*, vol. 36, 2010, pp. 317-344

[5] A. Malik, Z. Salcic, P. Roop: SystemJ Compilation using the Tandem Virtual Machine Approach, ACM Transactions on Design Automation of Electronic Systems, *14*, *(3)*, 2010

[6] A. Malik, Z. Salcic, A. Girault, A. Walker, S.C. Lee: A Customizable Multiprocessor for Globally Asynchronous Locally Synchronous Execution, Proceedings of Java Technologies for Real-time and Embedded Systems, Madrid, Spain, 2009, ACM

[7] M. Nadeem, M. Biglari-Abhari, Z. Salcic: RJOP – A Customized Java Processor for Reactive Embedded Systems, Proceedings of 48th Design and Automation Conference, San Diego, USA, 2011, ACM.

[8] G. Berry, "The semantics of pure Esterel," *Program Design Calculi*, vol. 118, 1993, p. 361–409

[9] [Hoare] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

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

Figure 4. GALS-JOP Data-path