JOP-Plus - A Processor for Efficient Execution of Java Programs Extended with GALS Concurrency

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

Abstract— In this paper we present an approach to efficiently mix Java with asynchronous and synchronous concurrency and execute it on a specialized Java processor extended with capabilities for concurrency and reactivity. A new processor, which uses JOP (Java Optimized Processor) as its base, executes concurrent programs that comply with Globally Asynchronous Locally Synchronous (GALS) formal model of computation by clearly distinguishing between concurrency and reactivity control flow and Java control flow. The new processor, called JOP-Plus, can be used for embedded and even real-time applications in which majority of code is written in Java and the overall programs specified and structured in SystemJ system-level concurrent programming language.

I. Introduction and Motivation

Java has not been widely used in embedded and real-time applications mainly due to two reasons: (1) its execution requires Java Virtual Machine (JVM) as an additional layer between the processor and Java byte codes and (2) automatic garbage collection introduces unpredictability element into response times of Java programs. In addition, Java's concurrency is completely non-deterministic and does not allow seamless composition of Java concurrent programs. Several attempts have been made to use Java in embedded and real-time applications by changing the language specification [1, 2], introducing more efficient and predictable garbage collection or by implementing byte-code execution in hardware (Java processors). A notable example of such processors is the open source JOP [3], which gives opportunity to modify and change the processor (instruction set, execution units) and explore more efficient use of Java in embedded and real-time world. Recently proposed system-level programming language SystemJ [4] extends Java with synchronous and asynchronous concurrency and reactivity, making it suitable for designing complex embedded programs. The language allows use of full Java and does recommend not using Java concurrency (threads), but relying on its own concurrency model based on formal Globally Asynchronous Locally Synchronous (GALS) model of computation (MoC). SystemJ programs can be deterministic and suitable for real-time applications if the continuous blocks of Java code embedded into those programs have bounded execution times. As Java programs which target JOP can be analyzed and their worst case execution times estimated, using JOP as the target of SystemJ compiler can result in more efficient execution of SystemJ compared to using traditional processors with JVM. However, implementation of concurrency and reactivity when using JOP results in complex and inefficient execution model, which maps on Java and is then compiled by Java compiler to target JOP. An attempt to extend JOP byte code repertoire and support SystemJ reactivity resulted in more efficient execution of SystemJ programs [5], but it did not address the issues of implementation of concurrency and overall program control flow. A further attempt is reported in [6], which builds on the ideas presented in [7], where a number of new native byte-codes are introduced to support more efficient implementation of SystemJ control flow.

This paper presents an integral attempt to address concurrency and reactivity of SystemJ by using existing JOP in a novel way and results in a new processor called JOP-Plus. The main idea is to separate the two components of the control flow in SystemJ programs, the one representing concurrency and reactivity control flow (CRCF) from the Java control flow (JCF) based on the original idea of tandem processing [7]. Then these control flows can be supported in the execution platform by using the extended JOP execution unit. This new execution unit has single instruction decoding unit and performs very efficient switching between CRCF and JCF when necessary. At the same time the compiler did not require any modifications. Thus, all SystemJ and JOP tools, without practically any modification, are made ready for the new processor. Cost-effectiveness of the approach is demonstrated on example programs. These are at the same time major contributions of the paper. The rest of the paper is organized as follows. Section II describes the background for the proposed approach and related works. Section III presents the execution flow of SystemJ programs in detail and its effect on JOP-Plus design. Section IV gives the details of JOP-Plus implementation. Section V presents the experiments and evaluation of JOP-Pluss performance. Finally, Section VI presents the conclusions and future work directions.

II. Background and Related Work

SystemJ programs comply with the GALS formal MoC. 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

978-1-4673-0772-7/12/$31.00 ©2012 IEEE

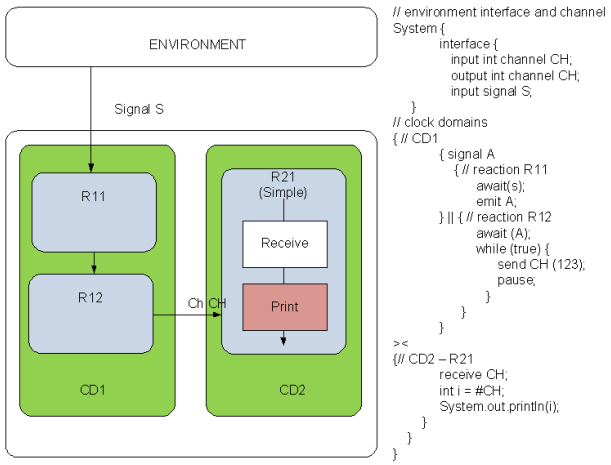

Fig. 1. An Example of SystemJ Program

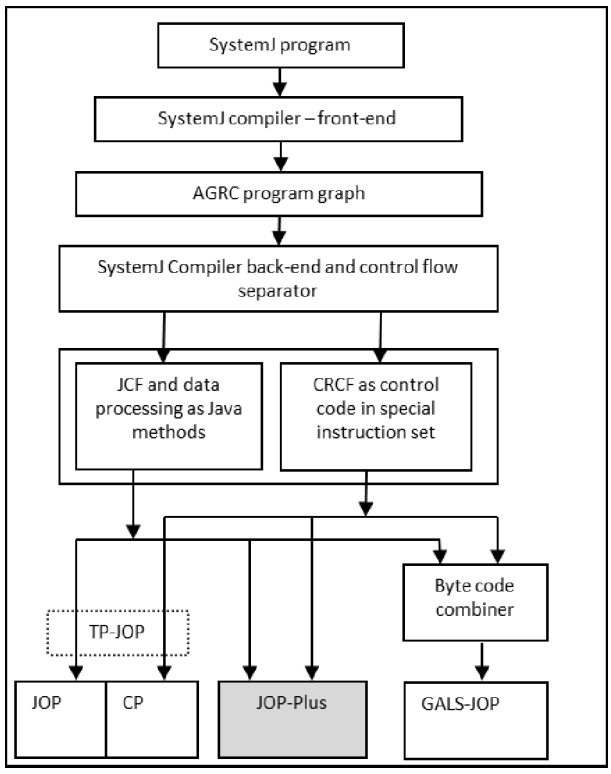

Fig. 2. Compilation and execution target strategies

concurrent processes, called reactions, which execute in lock-step, driven by a logical clock, called tick. 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 and comply with synchronous reactive MoC [8]. 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 [4, 9]. 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, and is referred to as concurrency and reactivity control flow (CRCF) in this paper. Instantaneous (Java) computations consist of continuous blocks of Java statements called Java action blocks or nodes (JAN), which are executed through the Java control flow (JCF). Thus, the SystemJ program execution can be in one of two major states: executing the CRCF or executing the JCF. While CRCF consumes logical time, the JCF is considered instantaneous.

An example of a simple SystemJ program that consists of two clock domains CD1 and CD2 is presented in Fig. 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.

Front-end of the SystemJ compiler [4] transforms a SystemJ program to an intermediate representation called Asynchronous Graph Code (AGRC) from which back-end of the compiler can target different execution platforms. Fig. 2 illustrates compilation and execution strategies of interest for all platforms that use JOP in some form as their execution target. JOP itself is a possible target, but resulting performance is fairly poor as it will be shown in Section V. Some improvement of performance was achieved with Reactive JOP, where reactivity is directly supported by hardware [5]. However, the separation of CRCF and JCF offers a larger space of SystemJ execution strategies. TP-JOP, based on the idea of tandem processor execution [4, 6] and GALS-JOP [6] indicate advantages of separation of control flows as they have different execution patterns and requirements. While, TP-JOP uses two processors (JOP and the control processor, CP) to implement the integrated control flow, the GALS-JOP extends JOP with a number of new byte codes to enable relatively efficient implementation and merging of the CRCF and JCF. The later requires some further modifications of the back-end of the compiler.

This work combines and extends the ideas of TP-JOP and GALS-JOP onto a new processor core JOP-Plus. The idea is to maintain separation of the CRCF and JCF by storing those two parts of SystemJ program in two different memories, and extend the execution capabilities of JOP with a number of new byte codes and new instructions, as well as with the micro-codes needed for their implementation. SystemJ program execution is guided by the CRCF, which in turn activates JCF whenever Java action nodes are executing and returns to the CRCF upon their completion.

III. Control Flow Execution

Java action nodes are implemented as Java methods, which in turn can call other Java methods. An example of the execu-

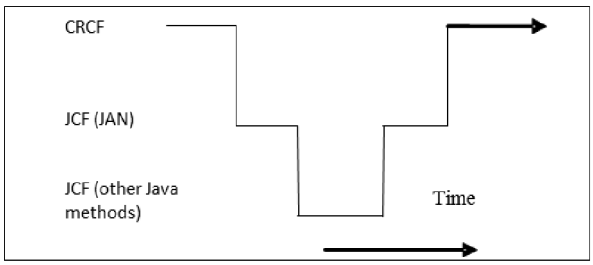

Fig. 3. Control flow of a SystemJ program

tion pattern is illustrated in Fig. 3. After initialization, which starts with loading the control and data part of the program in respective memories, Java main starts the execution and transfers control to the CRCF. When executing Java action nodes (JAN) in JCF, program can call other Java methods or return to the CRCF. These two transitions are implemented in different ways. While transitions between Java methods are using standard Java mechanisms (invocation and return), transitions between CRCF and JCF are performed with the support of hardware as it is explained in next section.

Mapping SystemJ programs onto CRCF and JCF is followed by the code generation which uses a special instruction set for CRCF and another set for JCF (standard Java byte codes) as described in [4]. Instead of fitting the special CRCF instructions into limited number of byte-codes as in [6], we preserve them and store them into two separate program memories in JOP-Plus, unlike using a single memory in the case of JOP. This way, instruction fetching has to be performed from a memory where CRCF or JCF is stored. The extended JOP execution unit is capable of decoding these instructions/byte-codes and executes them using micro-instructions stored in microcode ROM.

IV. JOP-Plus Architecture

In this section we describe the JOP-Plus architecture in details. A block diagram in Fig. 5 shows the new processor datapath and indicates additions and modifications of original JOP [10] (shaded).

A. Memory Organization

Besides memories illustrated in Fig. 4, JOP-Plus has a number of other memories used for specific purposes: stack cache, microcode ROM, and translation-table, which are all inherited from JOP and the register-file (RF), a small internal memory used for intermediate computations when executing CRCF instructions. The JOP-Plus has improved memory organization as compared to GALS-JOP and it does not need jump-table to implement jumps as the jump offset address in CRCF is part of the instruction.

The SystemJ program starts as Java program, which is compiled and initially loaded into the main memory. The rest of the main memory space is used as heap, which is used to store all Java objects and arrays, including the SystemJ valued signals and channels. The memory space used by inactive objects is reclaimed by garbage collector. Each JAN in JCF is

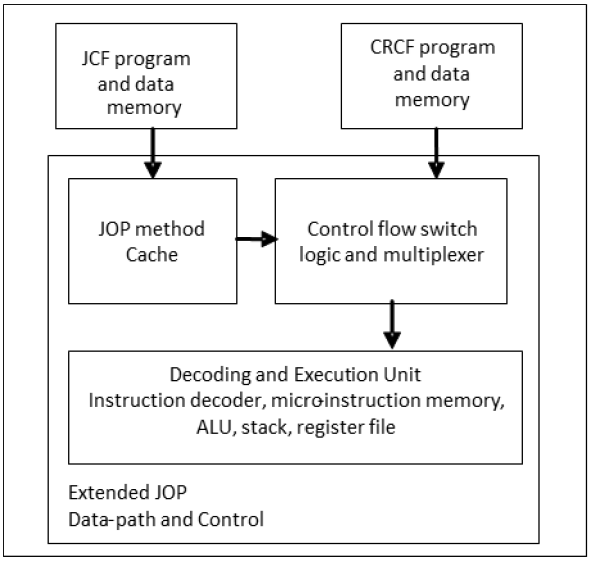

Fig. 4. Memory organization and program fetching

mapped to a separate Java method and is compiled to standard Java byte-codes. Most of the CRCF instructions are single 16-bit word, except the instructions which contain immediate operand, which require two words. The msb byte is the opcode of the instruction and the lsb byte (Rz & Rx) is address of the operands in the RF. These compiled CRCF instructions are initially enclosed in a Java array which is accessed by the main method.

The sole functionality of the main method is to initialize the CRCF program memory with contents of this array and shift the program flow to CRCF, which controls the execution and never returns to the main unless application has finished execution. The method cache serves as the instruction cache for JCF. The full code of a method is loaded into the cache before execution. The RF is used for temporary storage of the operands when executing CRCF. The CRCF data memory (DM) is 16-bit wide which contains the data structures for the CRCF, unlike the GALS-JOP [6] in which CRCF data structures are implemented as control arrays (CA) in the main memory, which is slow to access due to the limited bandwidth, resulting in slow execution of CRCF. It has been observed that CRCF DM is only a fraction of the CRCF program memory. Hence, the control data structures can be stored in the same memory as CRCF. The CRCF performs operations on the data structures. These operands are temporarily stored in a 16x16 RF. The translation-table contains the start addresses for the byte-code implementation in microcode. The original JOPs translation-table has 256 locations (all possible byte-codes). In the new implementation, translation-table is extended with new logical area where the op-codes of CRCF instructions reside. During CRCF execution, the corresponding entry at higher address in the translation-table is read. The microcodes which implement JVM and CRCF are in the same microcode ROM.

B. Processor architecture and instruction set

The CRCF has 33 distinct instructions and JCF requires 5 new byte-codes, compared to original JOP, to carry out functionalities such as CRCF program initialization and returning from the JAN method call as described in the next section. The new byte-codes corresponding to JCF are added to JOPs unused byte-code space, whereas the CRCF instruction op-codes are stored at higher address space in JOPs translation-table. Each byte-code and CRCF op-code is mapped to a single or a sequence of existing and 23 new microcode instructions.

In addition to the memories introduced in the previous section, the JOP-Plus architecture extends the JOP with a number of function specific registers and a hardware MAX unit. The address registers *mainptr* register and *crcfpc* register are used to store the main method structure address (during the startup) and the next CRCF instruction (when invoking a JCF method), respectively. The *mainptr* register is used to calculate the address of method structure being invoked. A temporary register T is used to transfer the data to and from A register, which is part of the JOP's stack. The hardware MAX unit is used to find out the maximum nibble value from the provided operands.

Depending on whether the processor is working in the CRCF or JCF mode, the CRCF instruction or byte-code is fetched from the CRCF program memory or the method cache in the byte-code fetch stage, respectively. Next, the byte-code is used as the address in the translation-table from which a start address of the microcode sequence of that particular byte-code is read. This address is used to fetch the microcode instruction from the microcode ROM. The instruction fetched during this (fetch) stage is fed to the decode stage for generating the control signals. The stack addresses are calculated during the same stage. In JOP, the operations are performed on top two elements of the stack stored in two discrete registers: TOS and TOS-1, labeled A and B. Each arithmetic/logical operation is performed with registers A and B as the source, and register A as the destination. This holds both for JCF and CRCF. In case of CRCF instruction execution, the RF is read and written in the same stage using microcode instructions.

All the JCF byte-codes are executed in the conventional way as in JOP. During the CRCF execution, the operands are pushed from register-file onto stack and the results are written back into the RF with data available at the top of stack, and register address is available as part of the CRCF instruction. Although the operands from the RF can be directly fed to ALU by-passing registers A and B, it will increase the propagation delay since the ALU falls in the critical path. During load/store of the data from the CRCF data memory (*load-immediate, load-direct, load-indirect, store immediate, store direct* and *store indirect*), the address is always provided in A, and data in B during a store operation. If the address/data is an immediate value, it is fetched from the CRCF program memory and pushed on the stack. All immediate arithmetic/logical (*and, or, add, subv* and *sub immediate*) instructions perform operation between the immediate value and content of register Rx and store the result into the register Rz except *sub* which does not store the result back into RF. The content of Rx register and operand values are pushed onto stack and the operation is performed. The result available on TOS is written back into register Rz pointed to by the instruction. In case of indirect instruction, operation is performed on the contents of registers Rx and Rz. The *chkend* instruction finds out the maximum of Rx and Rz3..0 nibbles and the result is stored in Rz. The contents of Rz and Rx are fed to a special hardware unit which finds the maximum nibble which is written to RF.

The jump instructions result in an unconditional jump to the target address provided as immediate (*jump-immediate*) value or the contents of a register (*jump-indirect*). The operand/register contents are put into A via T1 and stored into JPC, and the next instruction is fetched from the address pointed to by the immediate value. The *present* (jump if value is not present) and *sz* (jump if zero flag is set) CRCF instructions perform conditional jumps. The *switch* is a complex instruction and results in an un-conditional jump together with couple of memory read operations. It is used to decode the execution path in switch nodes in AGRC. The contents of register Rx are pushed on the stack and the memory location pointed to by its contents (parent node) is read into register T1. It contains the number of particular child node we want to switch to. This is added to parent node incremented by one (child nodes are stored next to parent nodes) to get the pointer to the selected child node in CRCF [4] and unconditional jump is performed.

The *datacall* instruction is used to directly invoke the desired data computation (JAN) presented as a method in JCF from within the CRCF instead of returning to main method and then invoking the desired method in conventional Java way. The JAN_ID is pushed on the stack. The address of main method structure stored in *mainptr* register is used to calculate the address of the structure of the desired method. All the methods in the JCF are ordered by their IDs which make it possible to calculate their structure address by knowing JAN_ID. The CRCF PC is saved in *crcfpc* register. The structure of the method, which is two words long, is read. The first word of method structure contains the information about the method code such as the start address and code length. This information is passed on to the memory subsystem which loads the desired method in the JOP's method cache and the execution starts. The second word contains the number of arguments, local variable count and constant pool address which are extracted and stored in appropriate registers. The *rtc* is a new custom byte-code added to support direct return to CRCF from JCF, without reloading method cache. It also writes the result of computation back into memory.

The result is provided as an argument available in A. The JAN_ID and data-lock position are provided in the RF. The register contents are pushed on the stack, the data-lock position is extracted and result is written to memory location pointed to by data-lock during return to CRCF. The description of the complete CRCF instruction set can be found in [4].

V. Experimental Results and Discussion

In this section, we present results of experiments conducted to evaluate and compare our proposed architecture to execute the SystemJ with a number of other approaches. We compare the execution of SystemJ on Altera general purpose embedded processor Nios II, tandem processor (TP) (NiosII + RECOP) [11], JOP [3] , TP-JOP [6], GALS-JOP [6] and JOP-Plus. Although we have several SystemJ benchmarks that can

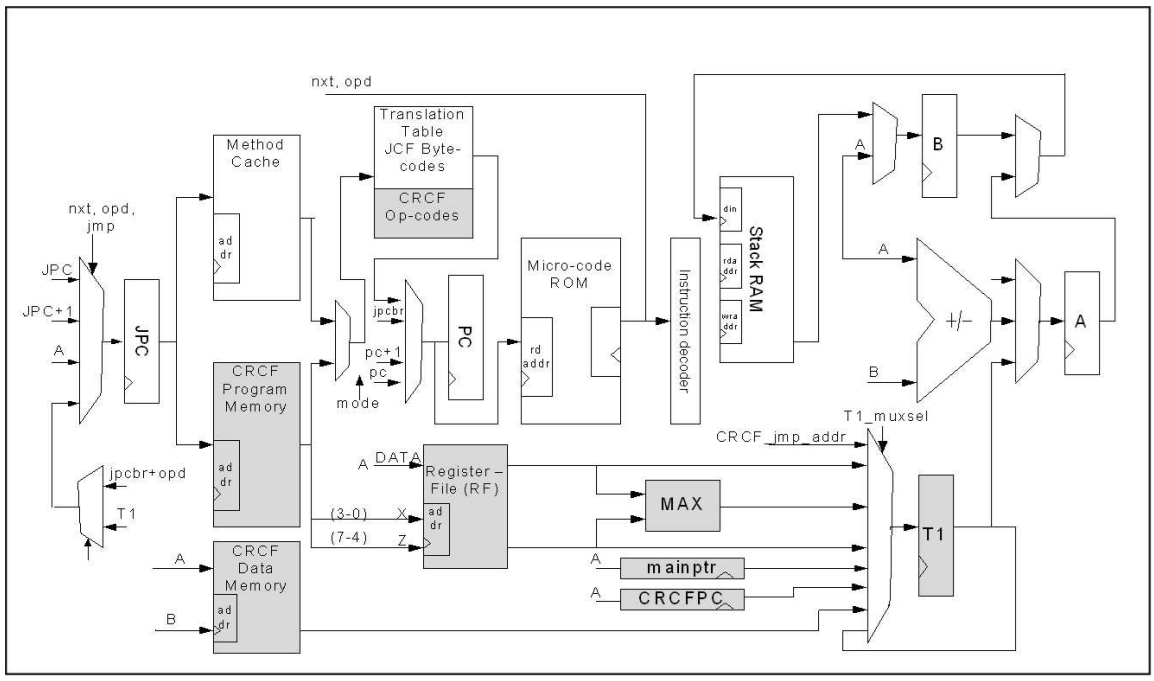

Fig. 5. JOP-Plus Datapath

be run on the JOP-Plus, we have selected only those which have been used in all other implementations. The benchmarks include both synchronous and asynchronous examples. The synchronous benchmarks include a runner's behavioral description (runner), a simple combination lock (cl) and demoloop (dl). The asynchronous case is represented by an asynchronous protocol stack (aps) example from [5]. Each of dl, runner and cl described in SystemJ consists of a single clock domain comprising of four reactions whereas aps has two clock domains and six reactions in total.

The execution speed comparisons are given in terms of the average execution time between two consecutive ticks for each clock domain, which are indications of the throughput of the new processor. Although JOP-Plus is capable of running at around 100 MHz, the results in Table I are given for 50 MHz for fair comparison with other existing approaches. The results show that JOP-Plus outperforms all the existing platforms. The JOP-Plus speed up is due to multiple factors. It outperforms TP-JOP, which uses the same instructions and executes them on a processor built specifically for that instruction set. Each instruction is executed in a constant time (three cycles except *switch* and *init* which take four cycles) in this case, while JOP-Plus has variable execution time for CRCF instructions. In case of CRCF execution only, TP-JOP will outperform JOP-Plus, but it loses the advantage in JCF execution and transfer of control between JCF and CRCF.

In TP-JOP, the two processors are running at the same time. The CP makes a call to DP (which is executing the main and is polling the DP) and receives the call and extracts the case number and clock domain fields. It calls a method consisting of clock domain code wrapped in a case statement. The corresponding clock domain invokes another method containing the JANs. The JOP has a special method cache which holds a

TABLE I
PERFORMANCE RESULTS IN TERMS OF TICK EXECUTION TIME IN MS

	GPP	**TP**	**JOP**	**TP -JOP**	**GALS -JOP**	**JOP -Plus**
dl	13	3.4	0.39	0.07	0.1	0.045
runner	520	6.9	0.64	0.15	0.13	0.104
cl	1000	2.5	1.66	0.141	0.16	0.106
aps CD0	0.8	0.3	0.14	0.05	0.06	0.044
aps CD1	3.15	0.4	0.25	0.13	0.16	0.12

complete method at a time. Invoking of a method and returning to a method results in cache miss and requires the cache to be loaded again, which incur huge cost. The higher call depth results in multiple loading of the cache on both invocation of methods and return from the method. Also, the data computation cases are enclosed in a single large method thus requiring more time to load it in the cache. On the other hand, the data computation in JOP-Plus is decomposed into methods corresponding to JANs, each having a small size. Secondly, it invokes the JANs methods directly from the CRCF without invoking main method, thus resulting in reduced call depth. The return to CRCF is direct without mediation of any other Java method, so no need to load caller method on return to the CRCF. The GALS-JOP [6] also has the similar characteristics but has more expensive invoke and return compared to JOP-Plus. GALS-JOP suffers from poor CRCF execution due to adopted mechanism where the CRCF instructions are translated to Java and all the operands and RF addresses are passed as arguments. As a result, each CRCF instruction is mapped to a custom byte-code to execute the specific functionality and number of byte-codes to push the arguments on the stack. In

TABLE II
RESOURCE (LE) USAGE IN DIFFERENT IMPLEMENTATIONS

	TP	TP-JOP	JOP	GALS-JOP	JOP-Plus
LEs	7350	6228	3579	4306	4250

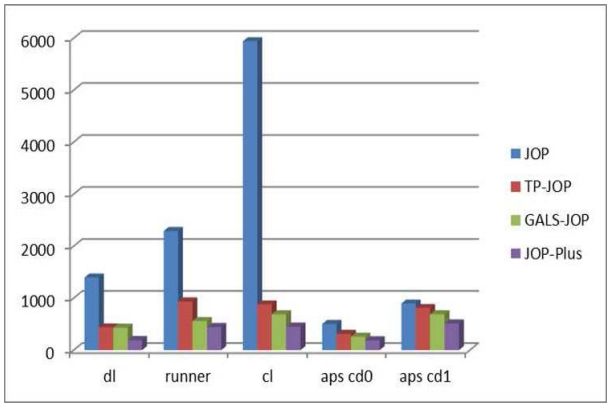

Fig. 6. Comparing the effectiveness of different approaches

JOP-Plus operand register addresses are available as part of the instruction. Another reason for the superior performance of JOP-Plus is that the local storage of frequently accessed CRCF data structures requires only one cycle to load and store.

Table II shows the resource utilization for different target platforms in terms of the logic elements (LE) used when implemented on an Altera Cyclone II FPGA. JOP-Plus implementation needs the addition of a few new microcode instructions but, the required storage limit (2Kx12, used by the JOP) will not be changed.

Comparison of effectiveness of different approaches is done by defining a performance measure that correlates Average Tick Times (ATT) with resource usage in terms of the number of logic elements (NLE) used for implementation. The performance indicator is specified as K*ATT*NLE, where K is a scaling constant, and shown in Fig. 6. The on-chip memory usage or each case is given in Table III. Note that only GALS-JOP approach utilizes jump table, which varies from 674 bytes to 2K bytes. It can be shown that the effectiveness of used memory for reducing the average tick times is very similar to the effects found for logic elements.

VI. SUMMARY AND CONCLUSIONS

In this paper, we presented a new execution platform for SystemJ language, called JOP-Plus. The SystemJ program is mapped to separate control flows, CRCF and JCF, and executed using only a single processor without any modifications to the existing compiler. This processor outperforms other execution platforms for SystemJ, and at the same time has the most efficient use of the FPGA real-estate which can be a very good candidate for embedded applications.

Having clearly established the superior performance of the JOP-Plus over the existing platforms, this is the time to focus on the scalability challenges. Future work includes the extension of the approach to multiprocessor/multi-core case, which requires exploration of communication between CDs and affinity of CDs (control and Java parts) to different cores. As CRCF execution times can be exactly calculated and JOP worst case execution times are also predictable, we plan to extend this work to the analysis of the worst case response times and optimization for use in real-time systems.

TABLE III
ON-CHIP MEMORY USAGE

	JOP	TP-JOP	GALS-JOP	JOP-Plus
Translation Table	254	254	280	291
Microcode Memory	2203	2203	2448	2804
Register file	-	32	32	32
FIFO	-	128	-	-
Jump Table	-	-	appl. dep.	-
Total	2457	2617	2760	3127

ACKNOWLEDGMENTS

The authors would like to thank Martin Schoeberl, HeeJong Park and Zhenmin Li for their help. Also, we appreciate the useful comments from the anonymous reviewers.

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 Theoretical 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, pp. 265-286, 2008.

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

[5] M. Nadeem, M. Biglari-Abhari, Z. Salcic, "RJOP - A customized Java processor for reactive embedded systems," *The 48th ACM/IEEE Design Automation Conference (DAC)*, pp. 1038-1043, 2011.

[6] M. Nadeem, M. Biglari-Abhari, Z. Salcic, "GALS-JOP - A Java embedded processor for GALS reactive programs," *accepted in Embedded-Com2011*, 2011.

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

[8] G. Berry, The semantics of pure Esterel, *Program Design Calculi*, vol. 118, pp. 361409, 1993.

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

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

[11] 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.