Microprocessors and Microsystems 64 (2019) 101–119

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Memory management of safety-critical hard real-time systems designed in SystemJ

Avinash Malik, HeeJong Park, Muhammad Nadeem*, Zoran Salcic

Department of Electrical and Computer Engineering, University of Auckland, New Zealand

ARTICLE INFO

Article history:
Received 11 July 2018
Revised 14 October 2018
Accepted 28 October 2018
Available online 1 November 2018

Keywords:
Compiler
Static analysis
WCET
SystemJ
Garbage collection

ABSTRACT

SystemJ is a programming language based on the *Globally Asynchronous Locally Synchronous* (GALS) *Model of Computation* (MoC) used to design safety critical hard real-time systems. SystemJ uses the Java programming language as the "host" language, for carrying out data computations, because Java provides clearly defined operational semantics, type and memory safety in the form of the *Garbage Collector* (GC), which help with formal functional verification. The same GC, which helps in functional verification, makes *Worst Case Reaction Time* (WCRT)[1] analysis challenging. Any WCRT analysis framework for GALS programs needs to consider the operations performed by the host language. It has been shown that the worst case time estimates for garbage collection cycles are in seconds, whereas the program's WCRT itself is in micro-seconds. These pessimistic estimates render the WCRT analysis framework ineffective. In order to overcome this problem, we develop a compiler assisted memory management technique for applications written in SystemJ. The SystemJ MoC plays the central role in the proposed technique. The SystemJ MoC allows clearly demarcating the state boundaries of the program, which in turn allows us to partition the heap, at compile time, into two distinct areas: (1) the memory area called the *permanent heap*, which holds objects that are alive throughout the life time of the application, and (2) the memory area used to hold all other objects, called the *transient heap*. The size of these memory areas are bounded statically. Furthermore, the memory allocation and reclaim procedures are simple load and pointer reset operations, respectively, which are guaranteed to complete within a bounded number of clock-cycles, thereby alleviating the need for large pessimistic WCRT bounds obtained due to the GC. Experimental results also show that the proposed approach is approximately three times faster, in terms of memory allocation times as compared to standard real-time GC approaches.

1. Introduction

Synchronous programming languages [1] allow building correct by construction safety critical hard real-time applications, because they are based on formal mathematical semantics, which allows the compiler itself to act as a model-checker [2] verifying functional properties of the application. The major benefit of the synchronous *Model of Computation* (MoC) is that it is based on the concept of a *Finite State Machine* (FSM). Languages such as Esterel [1] and its extension SystemJ [3] have been designed to reduce the state space explosion problem exhibited during formal verification, by adhering to a very strict programmer specified state demarcation approach. The programming model for these languages can be described as follows: a synchronous program is quiescent until one or more events occur from the environment. Upon detecting the event, the synchronous program *reactsinstantaneously* in zero time to respond to these inputs and becomes quiescent until the next input events arrive. The start and the end of this reaction transition demarcate states of the program — note that internal data updates do not lead to observable changes in the program state thereby reducing the state space explosion problem. Furthermore, the reaction being logically in zero-time is by definition *atomic* and always faster than the incoming input events, thereby guaranteeing that none of the incoming events are missed. In reality, the reaction does take time, one needs to find out the *Worst Case Reaction Time* (WCRT)[2] of any given synchronous program. This WCRT value determines the shortest inter-arrival time

* Corresponding author.
E-mail addresses: avinash.malik@auckland.ac.nz (A. Malik), muhammad.nadeem@auckland.ac.nz (M. Nadeem).

[1] Similar to *Worst Case Execution Time*.

[2] The Worst Case Reaction Time (WCRT) denotes time between two pause statements and it different from Worst Case Response Time which is the measure of time between inputs applied and output received. The Worst Case Response Time may comprise of a single or multiple Worst Case Reaction Times.

https://doi.org/10.1016/j.micpro.2018.10.007
0141-9331/

for input events from the environment. The requirement that all input events from the environment arrive at the best WCRT time units apart, guarantee that no input event goes unhandled and hence, enforces the *zero* delay synchronous hypothesis.

Techniques exist for computing WCRT estimates of synchronous programs providing support for data-computation via a non-managed language (primarily 'C') [4,5]. These WCRT estimation approaches suffer from a fundamental problem; they cannot incorporate complex data-computations within the WCRT analysis framework, because of the many undefined behaviors, type unsafety and the general lack of formal operational semantics inherent to low-level non-managed languages. Hence, in this paper we are interested in the WCRT analysis of programs that support data computation via managed-languages like Java, which provides a well defined operational semantics. SystemJ is the only programming language based on the GALS MoC that uses Java for its data computations. The key hurdle to the adoption of managed languages in the safety critical hard real-time application domain is *Garbage Collection* (GC) for automatic memory management, since *Worst Case Execution Time (WCET)* estimates for garbage collection cycles are very pessimistic [6]. However, by constraining Java within synchronous and GALS MoC, the opportunities for safe memory management for hard real-time systems become possible.

While static WCRT analysis is a must for real-time systems, it is not sufficient to guarantee the system safety, as the system can run out of memory during program execution. We identify the following requirements for WCRT analysis of SystemJ programs with automated memory management:

1. The number and clock-cycles of instructions used for memory allocation and reclaim should be statically bounded (to a constant value) at compile time. This is necessary, because the primary problem with long garbage collection times is the fact that garbage collection time depends upon the size of the physical memory and the number of objects allocated during program execution [7].
2. The size of each object and consequently the total memory consumption should be bounded (to a constant value) statically at compile time. JVM specification requires resetting all the memory (filling it with zeros) allocated for each object. Adhering to the first point requires that the size of the object be known statically, so that the loop zeroing the memory of the allocated object can be bounded. If all the object sizes can be determined statically, then the physical memory size can also be bounded.

The main **contribution** presented in this paper is a new memory management approach that is amenable to static WCRT analysis by satisfying the aforementioned requirements. More concretely our contributions are as follows:

- *Programming model inspired memory organization*: In this paper we present a new region based memory organization for programs developed in the SystemJ programming language.
- *Algorithmic analysis of memory consumption*: We develop the algorithm to statically analyse the *Worst Case Memory Consumption* (WCMC) of the data-computations performed using the host language Java within the larger SystemJ MoC.
- *Time analyzable back-end code generation and garbage collection*: We present a strategy to replace the non timing analyzable object allocation bytecodes (e.g., `new`) with real-time analyzable alternatives. Furthermore, we also present an object placement strategy, which allows for O(1) heap accesses in all cases.

Our technique works under the following assumptions:

- All loops in the programs, whether allocating memory or not, should be bounded statically — usually annotated by the programmer.

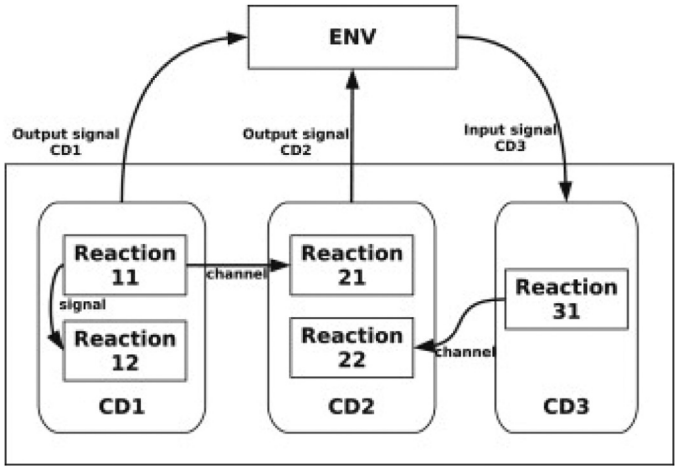


Fig. 1. A graphical representation of a SystemJ program.

- All recursions in the program should be bounded statically — usually annotated by the programmer.
- All memory allocations should be bounded statically — analyzed automatically by our algorithm.

These assumptions are necessary for approaches to worst case execution time analysis [8] and worst case memory consumption analysis [9], especially in the context of developing safety-critical real-time systems.

The rest of the paper is arranged as follows: Section 2 gives the background information needed to read the rest of the paper. Section 3 gives the motivating example to explain the SystemJ language and its WCRT analysis. Section 4 gives the overview of the new memory management scheme. Section 5 describes the main data-flow analysis algorithm used for compile time memory allocation. Section 6 explains the back-end code generation procedure. Section 7 gives the quantitative results comparing the presented technique with real-time garbage collection alternatives. Section 8 gives a thorough comparison of the work described in this paper with current state of the art. Finally, we conclude in Section 9.

2. Background

Before we present the key contributions of the paper, we dedicate this section to the description of the preliminary information needed by the reader.

2.1. The SystemJ programming language

SystemJ [3] extends the Java programming language by introducing both synchronous and asynchronous concurrency. SystemJ integrates the synchronous essence of the Esterel [1] language with the asynchronous concurrency of *Communicating Sequential Processes* (CSP) [10]. SystemJ is grounded on rigorous mathematical semantics and thus is amenable to formal verification. We will use the graphical representation of a SystemJ program in Fig. 1 for explaining the SystemJ MoC. On the top level, a SystemJ program comprises a set of clock-domains (CD1, CD2, CD3) executing asynchronously each to the other. Each clock-domain is a composition of one or more synchronous parallel reactions (Reaction11, Reaction12, etc), which are executed in lock-step with the *logical* tick of the clock-domain. Reactions within the same clock-domain exploit synchronous broadcast mechanism on objects called signals to communicate with each other. Reactions can themselves be composed of more synchronous parallel reactions,

SystemJ statements	Meaning
noop	do nothing
pause	complete a logical tick
[**input**][**output**][**type**]**signal** σ	declare signal σ
emit σ[(**value**)]	broadcast signal σ
abort (σ) **s**	preempt statement s if σ is true
suspend (σ) **s**	suspend statement s for one tick if σ is true
present (σ) **s1 else s2**	do s1 if σ is true else do s2
s1; **s2**	do s1 and then s2
s1||**s2**	do s1 and s2 in lockstep parallel
while(**true**) **s**	do s forever
jterm	Java data term
[**input**][**output**][**type**]**channel** ρ	declare channel ρ
send ρ[(**value**)]	Send a value to reaction in another CD
receive ρ	Synchronize and receive a value on channel ρ

Fig. 2. Syntax of the synchronous subset of the SystemJ language.

thereby forming a hierarchy. Finally, every reaction in the clock-domain communicates with the environment through a set of interface input and output signals. At the beginning of each clock-domain tick the input signals are captured, a reaction function processes these input signals and emits the output signals. These output signals are presented to the environment at the end of the clock-domain tick. Inter-clock-domain communication is achieved by implementing CSP style rendezvous mechanism through point-to-point unidirectional channels. A complete list of SystemJ kernel statements is shown in Fig. 2. The compile time memory allocation strategy described in this paper is applicable to each clock-domain individually and hence, in the rest of the paper we concentrate only on a single clock-domain. Also, any application described in SystemJ is a collection of clock-domains. These clock-domains are independent in terms of memory allocation and no memory is shared among them. The communication among clock-domains takes place through rendezvous. Therefore, local solution of memory allocation for a clock-domain is valid for the SystemJ program as well.

Signals, declared using the **signal** construct, are the primary means of communication between reactions of a clock-domain and between a clock-domain and its environment. A signal in SystemJ can be typed or untyped. Every untyped signal in SystemJ is called a pure signal and only consists of a Boolean status, which can be set to `true` or `false` via signal emission using the **emit** construct. A typed signal is called a valued signal, and consists of a value (of any Java data-type) in addition to the status. The value of a typed signal can also be set, optionally, via the **emit** statement. Once emitted, the status of a signal is `true` only in the next tick, but the value of a signal is persistent over ticks. Emission of a signal, broadcasts it across all reactions within the clock-domain.

SystemJ provides mechanisms to describe reactivity via statements such as **present**, **abort**, and **suspend**, which change the control-flow of a SystemJ program depending upon signal statuses. The control-flow of a program can also be altered via standard Java conditional constructs. Synchronous (lock-step) parallel execution of reactions within a SystemJ program is captured using the synchronous parallel (||) operator.[3] Furthermore, shared variable communication between synchronous parallel reactions is not allowed in SystemJ. The shared memory communication model is replaced by the signal emission based communication mechanism. State boundaries are demarcated explicitly by programmers using the **pause** construct. Finally, there are two types of loops in SystemJ: (1) temporal loops consisting of at least one **pause** statement, which execute forever and, (2) bounded data-loops like in standard Java. These data-loops are instantaneous and complete within the current tick.

2.2. The real-time execution architecture

Every static WCRT analysis technique needs intimate knowledge of the hardware that executes the program. For our purpose, we choose a dual-core time predictable execution platform called *Tandem-Java Optimized Processor* (TP-JOP) [11,12]. There are two main reasons we chose this processor architecture: (1) efficiency – it has been shown previously that the TP-JOP processor architecture is much more efficient for executing SystemJ programs compared to general purpose Java processors [13] and (2) there already exists tools for statically estimating tight WCRT bounds for synchronous SystemJ programs executing on the TP-JOP architecture [14].

Fig. 3 gives an overview of the TP-JOP architecture. The architecture consists of two time predictable cores. The first core is termed *Reactive Co-processor* (ReCOP) that executes the control flow instructions of a SystemJ program. The Java data computations are dispatched to the JOP core on a as needed basis by the ReCOP. ReCOP has a multi-cycle data-path, and each native ReCOP instruction is executed in 3 clock cycles. ReCOP core uses a single small private memory for both: data and program.

The JOP [15] core is a hardware implementation of the Java Virtual Machine (JVM). The JOP core is a four stage pipelined processor. Every Java bytecode is translated into one or more mi-

[3] Standard Java threading model is not allowed within SystemJ. Instead, synchronous and asynchronous parallel operators need to be used.

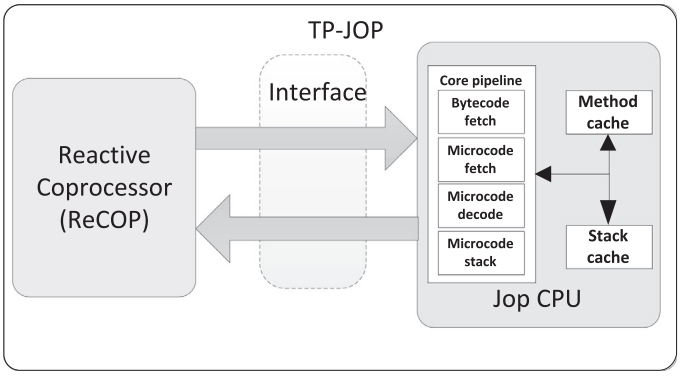


Fig. 3. Overview of the TP-JOP architecture.

crocodes, which are the native instructions of the JOP processor. The JOP pipeline has been designed to execute each microcode in one clock cycle and is guaranteed to be stall free. Furthermore, to guarantee time predictability, a novel cache architecture is implemented in JOP. There are two caches in JOP. The first is the stack cache [16] that acts as a replacement for the data cache found in general purpose processors. The second is the method cache [17] that acts as a replacement for program cache. A complete Java method is loaded into the method cache before execution and hence, there are only two program points: the *invoke* and the *return* bytecodes, which can lead to a method cache miss, which can be accounted for in the static WCRT analysis. Finally, the interface connecting the ReCOP and JOP has single clock cycle latency as described in [11].

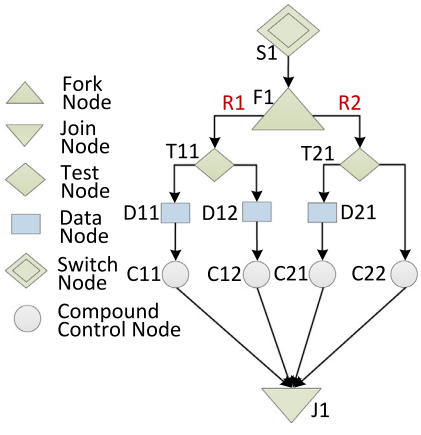
The overall program execution on the TP-JOP architecture proceeds as follows: the ReCOP leads the program execution since it executes the control-flow graph of the SystemJ program. When a data computation node is encountered, a call is dispatched to the JOP core with a method ID. Upon dispatch, the ReCOP itself stops processing further instructions until a result is returned from JOP. The JOP core polls continuously for an incoming request from ReCOP. Once a request is received, the JOP decodes the method ID and loads the method into the method cache. Once loaded, the requested method is executed and upon completion of execution, the result (a single bit) is returned back to ReCOP.

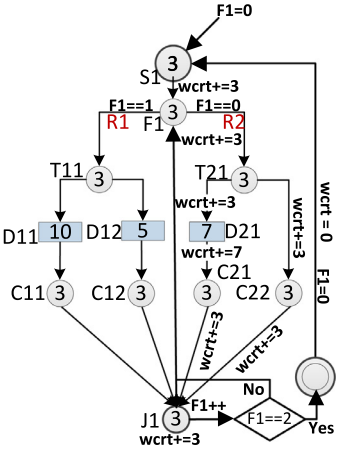
3. Static WCRT analysis of SystemJ clock-domains

In this section we use a pedagogical SystemJ synchronous program to describe the static analysis technique used to estimate the WCRT of a SystemJ clock-domain. The WCRT estimation technique is not the main contribution of this paper and hence, we only give a brief overview of the approach. The reader is referred to Li et al. [14] for a more detailed description of the technique.

Fig. 4a shows a simple SystemJ synchronous clock-domain. The clock-domain declares two input signals (lines 3 and 4) that are used to capture events generated from the external environment and one output signal (line 2) used to produce events back to the external environment. Two concurrent reactions (lines 6–16 and 18–26) execute in lock-step parallel, composed using the synchronous parallel operator (||) on line 17. Reaction R1 checks for an incoming event on the input signal `INPUT_A` (line 9). If an event is detected, and the value of the input signal is not zero, then a counter is incremented, else, it is decremented. This logic is carried out forever using an infinite loop (line 8). The pause construct (line 14) explicitly demarcates the end of program transition. Similarly, a counter `count_b` is incremented upon reception of an event on signal `INPUT_B` in reaction R2, else signal `OUTPUT` is emitted to the external environment.

```
1  SysJ_Example(
2    output signal OUTPUT;
3    input Integer signal INPUT_A;
4    input Integer signal INPUT_B;
5   )->{
6   {
7     int count_a = 0;
8     while(true){
9       present (INPUT_A) {
10        if (#INPUT_A == 0)
11        //...data computation
12        else ++count_a;
13      } else --count_a;
14      pause;
15    }
16  }
17  ||
18  {
19    int count_b = 0;
20    while(true){
21      present(INPUT_B)
22        ++count_b;
23      else emit OUTPUT;
24      pause;
25    }
26  }
27 }
```





(a) Synchronous SystemJ clock-domain (b) GRC of the clock-domain in Figure 4a (c) Uppaal representation of GRC

Fig. 4. Example SystemJ program with its GRC and Uppaal representations.

3.1. GRC representation of the clock-domain

The semantics of a synchronous program described in SystemJ is captured using an intermediate format called *GRaph Code* (GRC) [3]. The GRC of the SystemJ program shown in Fig. 4a is depicted in Fig. 4b. The GRC is a directed acyclic graph that explicitly describes the control flow of a synchronous program through primitive nodes.

Definition 1. *Clock-domain transition (or tick)*: A single clock-domain transition also called a clock-domain tick is a single traversal of the GRC from the source node (e.g., node S1 in Fig. 4b) to the sink node (e.g., node J1 in Fig. 4b).

Java statements in SystemJ are represented as data nodes. State encoding and selection are captured with enter and switch nodes, respectively. Present statements are denoted by test nodes. Concurrency embodied by parallel reactions is delineated using fork and join nodes. A fork node spawns parallel reactions, while a join node instructs that the parent reaction waits until all child reactions finish processing, which results in synchronized lock-step execution. Each node in the GRC is a schedulable unit, which can be either: (a) a Java data node (JDN) representing data-driven computation implemented as Java methods, or (b) a control node (CN) representing all nodes other than JDNs, implemented on the ReCOP. All control nodes between two data nodes are grouped together to form a single compound control node in order to simplify the GRC. For the GRC in Fig. 4b, the fork node F1 creates two parallel reactions R1 and R2 at the start of the program. The present statements (lines 9 and 21) are represented as nodes T11 and T21, respectively in the GRC. The Java data computations at lines 10-12, 13, and 22 are represented as nodes D11, D12, and D22, respectively.

3.2. WCRT estimation using the Uppaal model-checker

Finding the WCRT of a SystemJ clock-domain is equivalent to finding the longest execution path from the source node to the sink node in the GRC. Finding this longest path is non-trivial due to concurrent execution of the synchronous parallel reactions, the execution of different control branches depending upon the incoming input events from the external environment, and the state decoding within the switch nodes. There are a plethora of static WCRT estimation techniques in the research literature [5,18–21]. In this paper we describe the WCRT estimation technique based on model-checking. Any other static WCRT estimation technique can be used in our compiler framework. We use the Uppaal [22] model-checker to perform an exhaustive state space exploration of the GRC to find the WCRT of the SystemJ clock-domain. The exhaustive state space search carried out by the Uppaal model-checker gives WCRT estimates, for SystemJ programs, that are on average 30% tighter than those obtained by other approaches [14]. There are three steps involved in finding the WCRT of a clock-domain using Uppaal:

1. *Annotating the GRC nodes with WCET values*: Every GRC is annotated with timing information in order to compute the WCRT of the clock-domain. All CNs are executed on the ReCOP and hence, their timing information can be easily computed, as all native ReCOP instructions take 3 clock cycles to execute. The JDNs are dispatched, encapsulated within individual Java methods, for execution on JOP. One needs to compute the WCET for each of these JDNs, using the JOP provided worst case analysis tool (WCA) [8]. The computed worst case execution times for both CNs and JDNs are back annotated onto the GRC. The numerical annotations on the nodes in Fig. 4b show these back annotated WCETs.
2. *Translating the GRC into an Uppaal timed automaton*: Fig. 4b shows the GRC of the pedagogical SystemJ clock-domain captured as a *Timed Automaton* (TA) in Uppaal. All nodes and edges in the GRC are mapped as locations and edges in the TA. Given the TP-JOP architecture, we know that all control nodes are executed only on a single ReCOP. Consequently, all synchronous parallel reactions can only be executed sequentially. In order to enforce this sequential execution of the synchronous parallel reactions, we introduce extra integer type variables for each fork node, e.g., *F*1 in Fig. 4b. These fork variables are initialized to 0. Furthermore, a back-edge is added from the corresponding join node to the fork node, which increments the fork variable (*F*1 in Fig. 4b). Upon reaching the join-node, during state space exploration, the back-edge is taken as long as the value of the fork variable is not equal to the number of synchronous parallel reactions (2 in Fig. 4b), thereby guaranteeing sequential execution of the synchronous parallel reactions. An integer type variable *wcrt*, is added to the TA. [4] No additions are needed for the test nodes, since the model-checker implicitly backtracks and explores all branches of the test nodes. The *wcrt* variable is incremented by the WCET of individual nodes in the TA at every edge. Finally, to emulate the execution of multiple ticks, and consequently explore all branches of the switch nodes, a back-edge is also added from a dummy sink node (node with double circles in Fig. 4b) back to the source node. The *wcrt* and fork values are reset upon execution of this back-edge.
3. *Using a Computational Tree Logic (CTL) property to obtain the WCRT*: We model the WCRT estimation problem as verifying a CTL property in the Uppaal model-checker. The WCRT value lies in the bounded range denoted by $[WCRT_{lb}, WCRT_{ub}]$. $WCRT_{ub}$ is a safe upper bound on WCRT obtained by applying Max-Plus algebra [4] on the GRC, which is essentially summing up the maximum tick execution time of every reaction in the clock-domain. Similarly, we calculate a lower-bound of WCRT, termed $WCRT_{lb}$, by summing up the minimum tick execution time of every reaction in the clock-domain. After translating the GRC to a TA, we check the validity of the WCRT estimate of the clock-domain, termed $WCRT_{est}$, between $[WCRT_{lb}, WCRT_{ub}]$ by verifying a CTL property upon the TA using the Uppaal model checker. The CTL property is written as $A[](wcrt \leq WCRT_{est})$, meaning that the value of *wcrt* variable in the TA is less or equal to the $WCRT_{est}$ for every path in the TA starting from the initial location. We keep on reducing the value of $WCRT_{est}$ until the property does not hold on the TA. Upon violation of this property we are guaranteed that we have found the tightest $WCRT_{est}$ value.

4. A new memory management approach for SystemJ

The WCRT approach described in Section 3 is suitable *only* for programs that do not invoke the *Garbage Collector* (GC), since the GC execution time is unaccounted for in the aforementioned WCRT analysis framework. The main reason for this lack of GC incorporation is that the GC cycle time cannot be *practically* bounded; as the number of heap allocations in a Java program depends upon the application and during garbage collection a linked list of allocated objects may need to be traversed as one of the collection phases. Puffitsch [6] shows that a WCRT value can be obtained for programs, which invoke the GC, by pessimistically bounding the collection phase during static analysis. But, the resultant WCRT value is in seconds, orders of magnitude larger than the real execution time (usually in micro-seconds), which makes incorporating

[4] We use an integer variable instead of a clock variable in Uppaal, because we are computing the worst case reaction time in terms of clock cycles of the program, not in seconds.

GC difficult within the hard real-time scheduling framework. Our solution to the above problem is to eliminate garbage collection altogether. More concretely, we divide the heap space into two parts: a bounded permanent heap and a transient heap, which are time and space bounded and efficient.

Signals (valued or pure) are the only communication primitive within a SystemJ clock-domain. Signals, internally themselves implemented as Java objects, are alive throughout the lifetime of the application. The proposed heap organization is based on one **key insight**; there are two types of Java objects in a SystemJ clock-domain:

- *Permanent objects*: Java objects that are emitted via signals. These objects like signals are alive throughout the application lifetime.
- *Transient objects*: Java objects that are created internally for computation, but are never emitted via signals. Transient objects are only alive during a single clock-domain tick transition and can be deallocated at the end of the tick transition.

The basic idea is to allocate the permanent objects, we consider signals and the values associated with them to be permanent objects as well, within a special memory area called the *permanent heap* and allocate all transient objects within a *transient heap*. The transient heap gets reclaimed at the end of every clock-domain tick transition, by simply resetting the allocation pointer to the start of the transient heap space. With this change, the proposed runtime Java memory organization is compared with the original real-time GC based memory organization in Fig. 5.

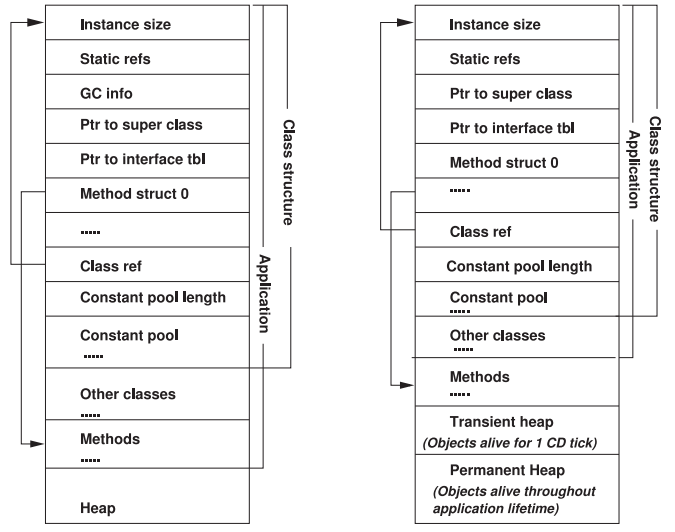
Both runtime memory organizations; original (Fig. 5a) and proposed (Fig. 5b), consist of a number of common elements required to execute every Java application such as, the constant pool, method structure table for instance method invocation, references to static objects, etc. The major point of difference is that the *GC info* word, which describes characteristics of the collector itself, does not exist in the proposed runtime memory organization and the *heap* space is divided into two parts; the transient heap and the permanent heap. Looking at the proposed runtime memory organization two requirements become very important:

1. There should be no pointer pointing from the permanent heap to the transient heap, else an application might terminate at runtime with a `NullPointer` exception – since the transient heap may be reclaimed or overwritten at the end of the clock-domain transition. Moreover, in the much worse scenario, an incorrect value might be read, in which case the application will be functionally incorrect. Both these scenarios are unacceptable for safety-critical systems.
2. The permanent heap is never reclaimed and hence, permanent heap space should be judiciously allocated. In fact, we need to bound the permanent heap space at compile time.

In the rest of the paper we describe compiler transformations that accomplish the proposed memory management technique.

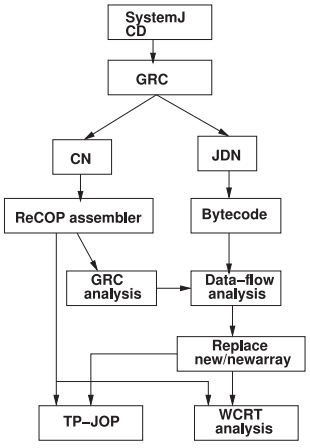
5. Static analysis for compile time memory allocation

The complete compiler tool-chain flow is shown in Fig. 5c. The SystemJ clock-domain is first compiled into the intermediate GRC format. Next, the control nodes (CN) and the Java data nodes (JDN) are split and compiled separately into ReCOP native assembly and Java bytecode, respectively, for execution on the TP-JOP platform. Two types of compiler analysis are then performed on the produced native ReCOP assembler and Java bytecode, respectively: (1) GRC analysis is used to determine the *must* and *may* Java reachable methods from any given Java program point and (2) *forward* and *backward* data-flow analysis is performed to find the objects and arrays that need to be allocated to the permanent heap. Finally,



(a) Original JOP runtime memory organization

(b) Proposed JOP runtime memory organization



(c) The tool-chain flow

Fig. 5. The tool-chain flow along with the Java runtime memory organization in JOP, before and after transformation. The Java stack is on separate physical memory as shown in Fig. 3.

these object allocation bytecodes need to be replaced by their time predictable alternatives.

The need for these two types of analysis can be explained using the simple example in Fig. 6a. This SystemJ code snippet declares a valued signal `S` (line 1), and an integer variable `a` (line 2). Variable `a` is incremented (line 3) and emitted via signal `S` (line 4). After emission, the tick expires as indicated by the `pause` statement. In the second clock-domain transition, variable `a` is initialized again, since in SystemJ the only values that are persistent across ticks are signal values. Variable `a` is first initialized to the value held in signal `S` (from the previous tick) and then incremented (lines 10–11). The result of this increment is emitted via signal `S`. Two methods are produced in the back-end Java code for execution on JOP (Fig. 6b). The first method `MethodCall1_0` (lines 27–31) initializes `a` and then increments it. Finally, it sets the value of the signal `S` as the *object* `a`. The second method `MethodCall2_0` (lines 32–36) increments the variable `a` and changes the object reference of signal `S`.

Our objective is to find out all the *program-points* that allocate a new object, whose returned reference is set as a signal value via

```
1  int signal S;
2  int a = 0;
3  a = a + 1;
4  emit S(a);
5  pause;
6  /*Variable values can
7   only be carried
8   over tick boundaries
9   via signals */
10 a = #S;
11 a = a + 1;
12 emit S(a);
```

(a) SystemJ code snippet of a single sequential reaction

```
1  public class FruitSorterController {
2   //signal S declaration
3   private static Signal S;
4   private static int a;
5   public static void init () {
6    S = new Signal();
7   }
8   public static void main(String args){
9    init();
10   while(true){
11    //poll on the method call request from ReCOP
12    int methodnum = Native.rd(Const.METHOD_NUM);
13    switch(methodnum){
14     case 0: ...
15     ...
16     //Call method for
17     //lines 2 - 4 (Figure~6a)
18     case 1: Native.wr(RESULT_REG,
19     MethodCall1_0());
20     //Call method for
21     //lines 11 - 12 (Figure 6a)
22     case 2: Native.wr(RESULT_REG,
23     MethodCall2_0());
24    }
25   }
26  }
27  private static boolean MethodCall1_0(){
28    a = 0;//line 2, Figure 6a
29    a = a + 1;//line 3, Figure 6a
30    S.setValue(new Integer(a));
31  }
32  private static boolean MethodCall2_0(){
33    a = S.getPreValue(); //get value from previous tick via S
34    a = a + 1;
35    S.setValue(new Integer(a));
36  }
37 }
```

(b) Produced Java code

Fig. 6. Example showing the need for GRC and data-flow analysis of a SystemJ program.

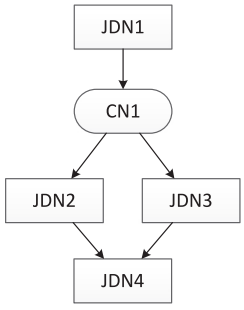
the `setValue` virtual method call on the signal object. We perform data-flow analysis (ReachDef analysis [23] to be exact) to find out such program points in the Java program generated for execution on JOP (e.g., Fig. 6b). Fields or method local object references might be set as signal values and hence, our data-flow analysis is a context and flow-sensitive intra-procedural and inter-procedural analysis, i.e., the data-flow analysis not only examines each Java method, but also traverses the call-graph of the generated Java program.

A call-graph generated from *just* the Java program is incomplete, since method calls that *must happen before* a given program point *appear* as *may happen before* a given program point. Consider the SystemJ program in Fig. 6a, we know for sure that program point at line 3 *must* be executed before program point at line 11. But, in the generated Java program (Fig. 6b), the sequential control-flow of the *SystemJ program* has been turned into branching control-flow (lines 13–24). Just looking at the Java program, one can only state with certainty that method `MethodCall1_0` *may* be called before method `MethodCall2_0`. More disconcertingly, just looking at the Java program, one can even say that method `MethodCall2_0` may be called before `MethodCall1_0`, since the dependence edge, which is clear in the SystemJ program, is lost in the generated Java program. The GRC analysis step produces the may and must method callers for any callee in the generated Java code. Note that we cannot produce a single Java method call coalescing the two Java methods: `MethodCall1_0` and `MethodCall2_0`, because there is an intertwined control construct, `pause`, which needs to be executed on ReCOP. We describe these GRC and the data-flow analysis steps in the following sections.

5.1. GRC analysis

GRC is a *directed graph* $G = (V, E)$, where V is the set of vertices (nodes of GRC) and E is the set of ordered pairs of vertices. Each edge $e = (v_i, v_j), \forall e \in E, v_i \in V, v_j \in V$ is a tuple denoting that the edge is directed from v_i to v_j. Then a function $\lambda : v \rightarrow E_i,\ E_i \subseteq E$, maps *each* vertex to a set of edge(s) where $\forall (v_i, v) \in E_i$ and E_i may be $\emptyset$. When $|E_i| = 1$ we say v_i *must* happen before v whereas when $|E_i| > 1$ we say any $v_i \in E_i$ *may* happen before v.

In this section, we illustrate how the data-structure, which contains information on the *may* and *must* relationships between



(a) An abstracted GRC graph

```
1  (mList
2   (mNode JDN1 (Must Null) (May Null))
3   (mNode JDN2
4    (Must (mNode JDN1 (Must Null) (May Null)))
5    (May Null))
6   (mNode JDN3
7    (Must (mNode JDN1 (Must Null) (May Null)))
8    (May Null))
9   (mNode JDN4 (Must Null)
10   (May
11    (mNode JDN2
12     (Must (mNode JDN1 (Must Null) (May Null)))
13     (May Null))
14    (mNode JDN3
15     (Must (mNode JDN1 (Must Null) (May Null)))
16     (May Null))))
```

(b) An output of the GRC analysis

Fig. 7. An example of GRC analysis.

the nodes, is obtained from the GRC. Consider a snippet of GRC graph shown in Fig. 7a. Here, the root node, JDN1, has no parent whereas it has an immediate child CN1. CN1 has two children, JDN2 and JDN3, and both of these JDNs have the same child JDN4. This graph when given as an input to our GRC analysis tool, returns the result as a S-expression as shown in Fig. 7b. The resultant inductive data-structure consists of a set of nodes called `mNode`, which has three fields: (1) the node name of type string, (2) `Must` field of type `mNode` and (3) `May` field, which is a set of type `mNode`. This data-structure can be used to identify potential callers of each JDN in the GRC graph. For example, since JDN1 has no parents, its `Must` and `May` fields are both `Null` (line 2). On the other hand, JDN1 must be called before JDN2 or JDN3, hence `Must` of both JDN2 and JDN3 is JDN1 (lines 4 and 7). Lastly, JDN4 has two parent nodes; JDN2 and JDN3. Therefore, its `May` field has both JDN2 and JDN3 (lines 11–16). It should be noted that we are only interested in callers of JDNs, hence CNs are not included in the final result (Fig. 7b).

5.2. Data-flow analysis

Given the class file(s), the data-flow analysis carried out is shown in the flow-chart in Fig. 8. Being a complex procedure, we use simple code examples to describe the data-flow analysis. We divide the presentation of our data-flow analysis into two sections. Section 5.3 presents the *Intra-procedural* data-flow analysis, which is complemented with the *Inter-procedural* data-flow analysis in Section 5.4.

5.3. Intra-procedural analysis

Fig. 9a shows a Java program as a call-graph. There are three methods: `main`, `init`, and `MCall`. The `main` method calls the `init` and the `MCall` methods sequentially. The `init` method initializes a static object reference `S` of type `Signal`. Fig. 9b shows the control-flow-graph (CFG) of the `MCall` method. It consists of four basic blocks annotated as BB1, BB2, BB3, and BB4, respectively. BB1 initializes objects `ob` and `oc`, whose types `b` and `c` are inherited from a single parent class `a` as shown in Fig. 10a. Object `oa` (BB1, line 3) of type `a` is initialized to either type `b` or `c` (BB2, line 2, BB3, line 3) depending upon the value of the Boolean `a_thread`. Finally, object `oa` is emitted via signal `S` (BB4, line 1).

- Step1 of the data-flow analysis algorithm starts by scanning the Java program for program points allocating signals (e.g., method `init` in Fig. 9a). A globally accessible set *pp* is defined, which holds all the program points allocating objects that will be placed in the permanent heap. Step1 scans the call graph from the starting program point (usually the `main` method) for signal allocation and places these program points (including the one in `main`) into the set *pp*. Each individual tuple within the set *pp* consists of one or more program points that allocate objects. After the scan phase, the set *pp* holds $\{(5, \{\texttt{init} : \texttt{BB1} : 1\})\}$ for the example in Fig. 9a. The first element of the tuple gives the *instance* size of the `Signal` class, the second element is the allocation program point [5] for the example in Fig. 9a. If more than one signal is allocated, all these program points are placed in the set *pp*. Finally, Step2 is called to find all the signal emission program points.
- Step2 is a recursive procedure that scans all reachable methods, in the call-graph, for the `setValue` virtual method call on the signal objects. These program points are placed in the set *ppc* for each reachable method individually. For the example in Fig. 9, this involves scanning method `MCall` and placing the program point: 'BB4:line 1', in set *ppc*. Finally, Step3 of the algorithm is called to obtain all object allocation program points whose returned object references are emitted via signals.
- Step3 analyzes the emission expression (`setValue` method) for every program point in set *ppc*. An emitting expression can *only* emit a method local variable or a field. Either Step4 or Step5 is called, depending on whether a field or a local variable is emitted (or formally called affected), respectively. In our

[5] The allocation program point is actually at the bytecode level, but in this paper we keep the program points at the Java source code level for ease of understanding.

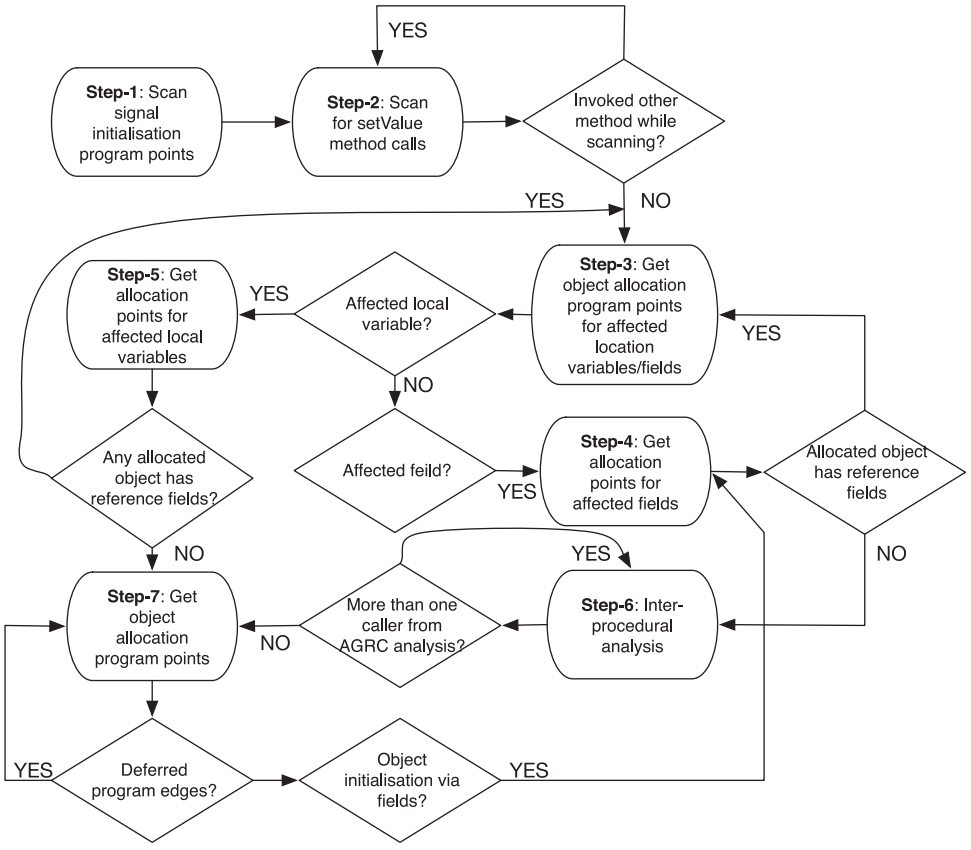
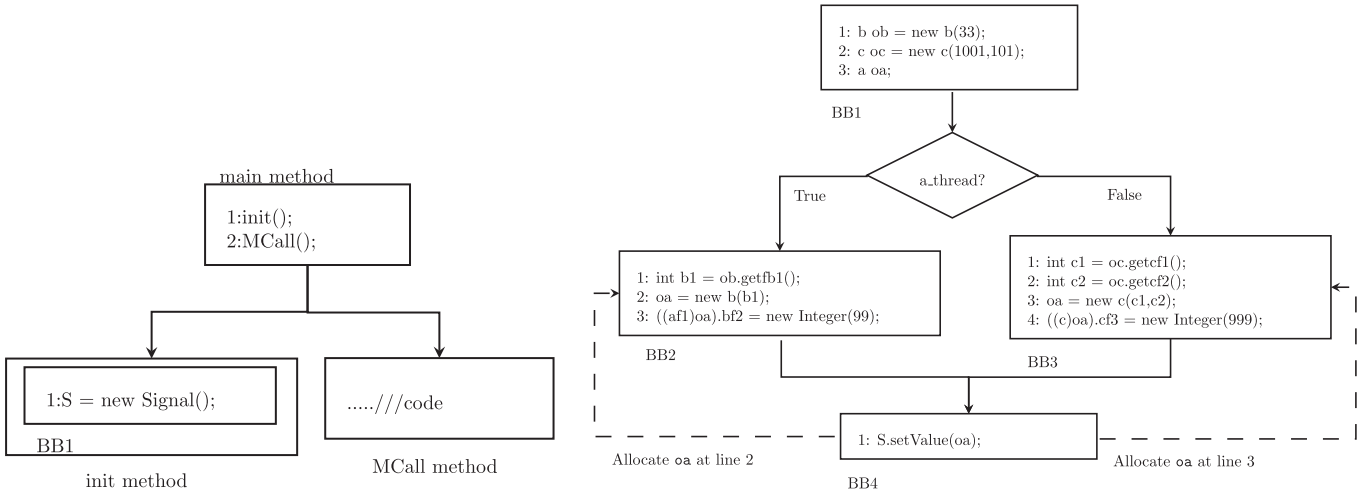


Fig. 8. The flow-chart for the data-flow analysis algorithm.

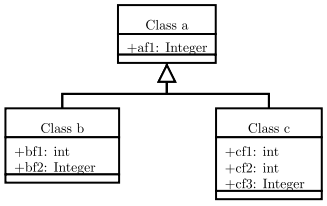


(a) The call-graph for the running example (b) The control-flow graph of the MCall method

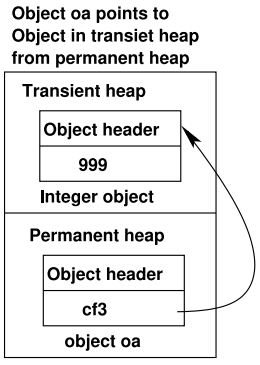
Fig. 9. Running example used to explain the intra-procedural data-flow analysis.

current example a local variable (`oa`) in Fig. 9b, BB4, line 1, is being affected and hence, Step5 is called.

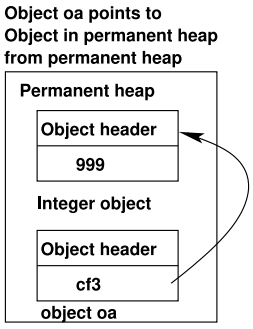
- Step5 extracts the variable from the affected expression and calls Step7 for analysis. Step7 is the core of the whole data-flow analysis procedure that performs reachability analysis to identify the program points initializing the objects to be placed in the permanent heap. Before proceeding to Step7, let us assume that we have already obtained the correct program points from Step7 in set *alloc_pps*. For our current example, given that the emitting program point is 'BB4:line 1', *alloc_pps* = {`MCall : BB2 : 2, MCall : BB3 : 3`}. Looking at Fig. 9b, it is clear that object reference `oa` can be initialized at either in BB2 line 2 or BB3 line 3, depending upon the value of variable `a_thread` during program execution. For each of these program points, Step5 first performs program analysis to guarantee that the object reference does not *escape* the lifetime of



(a) UML class hierarchy for classes `a`, `b` and `c` used in Figure 9b



(b) Pointers from permanent heap to transient heap



(c) Points-to only in permanent heap

Fig. 10. The UML representation of the class hierarchy used in Fig. 9 and the points-to problem.

the method [24].[6] If the object reference does not escape the lifetime of the method, then the algorithm checks if there are any reference fields within this object. If so, the algorithm returns back to Step3, for migrating all object allocations, whose returned reference is held in these fields, into the permanent heap.

Consider the program point in BB3 at line 3. Object `oa` is initialized as class `c` (for program point in BB2 line 2, `oa` is initialized and analyzed as class `b`), which has three fields (c.f. Fig. 10a): two (`cf1` and `cf2`) are primitive fields, but the third: `cf3` is a reference field of type `Integer`. The `cf3` reference field is initialized to an `Integer` object in BB3 line 4. Fig. 10b shows the address held in `cf3` if Step3 is not called from Step5. Reference field `cf3` points to an object in transient heap space, which violates the first requirement in Section 4. Calling Step3, guarantees that all potential *pointed-to* objects from the permanent heap are also migrated to the permanent-heap. Note that Step3 and Step5 are mutually recursive and hence, perform a chained permanent heap migration. For the running example, the result of calling Step3 from Step5 is shown in Fig. 10c.

The final computation that Step5 performs is computing the size, which will be reserved in the permanent heap, of object `oa`, given that *alloc_pps* may have multiple (two in our current example) elements. The primary idea is to reserve the maximum size from amongst all the different types being initialized. For the running example, `oa` can be of type `b` or `c` during program execution. We compute the *instance* size of both these types as 3 and 4 words, respectively[7] and reserve 4 words (plus object header size) in the permanent heap space. Step7 returns multiple program points in set *alloc_pps* if and only if these program points are mutually unreachable. This, guarantees that the reserved permanent heap space can be *reused* at runtime by all program points in set *alloc_pps*.

- **Design decision**: We have disallowed method local object references from escaping as a trade-off between space utilization and functional correctness. If we allow, for example `oa` in Fig. 9b, to escape method `MCall`, then any of the reference fields of `oa` (e.g., `cf3`) might be reinitialized at some other program point. This new program point would also then need to be considered and space on the permanent heap would need to be reserved for the object allocation. This can result in the permanent heap becoming too large. Conversely, if the escape analysis is not performed, then there will be pointers pointing from the permanent heap to transient heap, which may result in potentially incorrect program behavior. Simply not allowing any object reference to exceed the lifetime of the method avoids both these problems.

– Step7, given a program point that affects a variable (also termed the *use* program point) gives one or more program points initializing that variable (also called the defining program point). Step7 first builds the standard *use-def* chains [23] to obtain the program points defining the variable. If the defining program points are `new` or `newarray` bytecodes [25], then these program points are simply unioned into the set of program points that will be returned by this function. If the defining program points are themselves affecting variables, then Step7 is called recursively to follow the so called deferred program edges [26] to finally reach one or more program points that allocates the object or if the variable is not initialized then gives a compile time `not initialized` error. Note, that this amounts to carrying out a complete alias analysis. If the defining program point affects a field, then Step4 is called, which calls Step7 via Step6, we describe Step4 and Step6 in the next section. If the defining program point is anything but any of the aforementioned statements, then an error is raised, stating that the variable might be initialized outside the method being analyzed. Finally, Step7 makes sure that the program points being returned are not reachable from each other.

- **Design decision**: We have consciously decided to not allow object allocation outside the method that uses that object in Step7. The objective of our data-flow analysis is to identify all the program points that allocate objects whose returned references are emitted as signal values. We are performing compile time memory allocation, which, as we will see later, requires us to replace the `new`, `newarray`, etc, bytecodes with a different set of bytecodes (c.f. Section 6). We cannot blindly replace these bytecodes in methods other than the ones being analyzed, because, consequently *any* other program point, related or unrelated to signals, using the same object initialization program point would overwrite the object value as it inadvertently shares the same object. A simple example elucidating the situation is shown in Fig. 11.

  In Fig. 11a, we initialize two `Integer` object type variables; `A` and `B` to constant `int` values. Then we emit `A` via signal `S`. The bytecode produced from this Java program is shown in

[6] Let *O* be an object reference and *M* be a method invocation. *O* is said to escape *M*, if the lifetime of *O* may exceed the lifetime of *M*.

[7] According to Java semantics, all fields of a parent class are inherited by the children classes and hence, the sizes for `b` and `c` are 3 and 4 words, respectively.

```
1  private static Signal S;
2  public static void main(String args){
3   Integer A = 0;
4   Integer B = 1;
5   S.setValue(A);
6  }
```

(a) Example Java code snippet

```
1  iconst_0
2  invokestatic  #2// valueOf:(I)Ljava/lang/Integer;
3  astore_1
4  iconst_1
5  invokestatic  #2// valueOf:(I)Ljava/lang/Integer;
6  astore_2
7  return
```

(b) Produced Java bytecode

```
1  new #3 // class java/lang/Integer
2  dup
3  iload_0
4  invokespecial #10 // Method "<init>":(I)V
5  areturn
```

(c) The bytecode for method `valueOf` in Integer class

Fig. 11. Inadvertent object sharing problem.

Fig. 11b. First, the constant 0 is loaded onto the stack and the method `valueOf` in the `Integer` class, from the Java standard library, is invoked, which allocates memory and initializes the returned reference field to 0, the returned reference is then stored on the stack (lines 1–3), same is done for the object `B` in lines 4–6. The actual object initialization is carried out inside the `valueOf` method, Fig. 11c, line 1. If we were to replace this `new` bytecode, to point to some memory words in the permanent heap, then both; `A` and `B` will point to the same place in the permanent heap. Consequently, first a constant 0 will be written in the allocated memory (Fig. 11b, lines 1–3) and then this value will be overwritten to 1 (Fig. 11b, line 5), resulting in functionally incorrect code. The *only* solution to this problem is to *inline* the called method, `valueOf` in this case. But, extreme caution is required when in-lining methods, because the method might be too large to fit in the method cache, or more than one such methods might need to be in-lined (in case a method itself calls another method, which does the actual object allocation and so on and so forth). To put the cache size restriction into perspective, our benchmark execution platform has a very limited cache size of only 1KB. Thus, we have decided to disallow object allocation outside of the method being analyzed. As a consequence of this design decision, the SystemJ programmer now needs to make explicit deep copies as shown in Fig. 9b BB2 line 3 and BB3 line 4.

5.4. Inter-procedural analysis

Until now we have only described those parts of our data-flow analysis that handle local method references being emitted via signals. Step4 and Step6 (c.f. Fig. 8) described in this section work in conjunction with the previously described steps in order to handle cases where Java field references may be affected by signal emissions. We again use a simple example shown in Fig. 12 to describe Step4 and Step6 of our data-flow analysis.

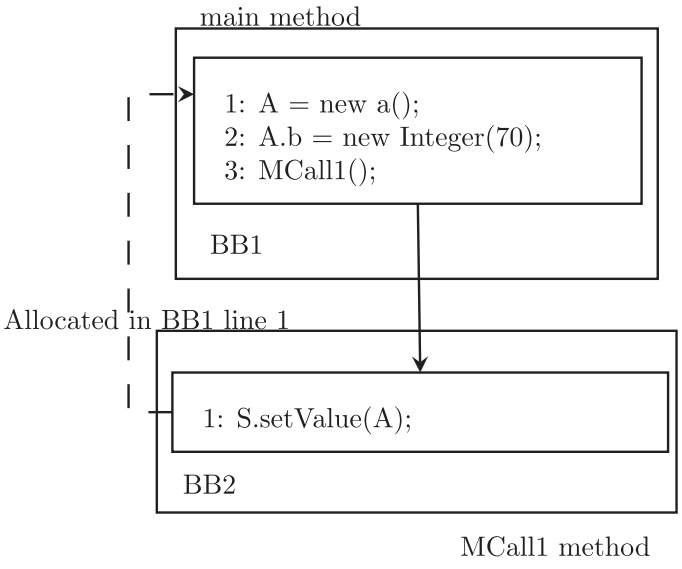


Fig. 12. Inter-procedural analysis with fields. `A` and `S` are static fields.

- Step4 is the dual of Step5. It works on Java fields rather than method local variables like Step5. The other difference between the two is that Step4 calls Step7 via Step6, which performs the inter-procedural analysis.
- Step6 can be conceptually partitioned into two parts. Part-1 returns the program points allocating objects within the same method that the field is being used. The more interesting part is part-2, which carries out *inter-procedural* analysis if the object containing the affected field being analyzed is not allocated in the same method.

Let us consider the Java program call-graph in Fig. 12 to explain the inter-procedural analysis. The `main` method initializes two objects: `A`, a static field, of type `a` (c.f. Fig. 10a) and

```
Input: maxMem: Maximum memory
       address
/* defs is the same set as pp in
Figure 8                          */
Input: defs: The program points to
       replace
Input: program: The program
1  let allocPtr ← maxMem;
2  let header_size ← 2;
3  foreach (size, dd) ∈ defs do
4      let dd ← sortUniqueDescending (dd);
5      let M ← load (dd);
6      let allocPtr ←
       allocPtr − (size + header_size);
7      if allocPtr < 0 then raise No_mem;
8      foreach d ∈ dd do
9          replaceByteCode
           (allocPtr, d, M, size);
10     end
11 end
```

(a) Bump-pointer memory allocation

```
Input: allocPtr: allocation pointer
Input: pp: program point to replace
Input: M: Method containing the
       program point
Input: size: Size of the allocated
       object/array
1  if pp = new then
2      let arg ← getArg (pp);
3      replaceNew (pp,M,arg,size);
4      adjustConditionals (M,31);
5  else if pp = newarray then
6      let arg ← getArg (pp);
7      if arg = primitive then
8          replaceNewArrayB
           (pp,M,arg,size);
9          adjustConditionals (M,32);
10     else if arg = single − dimension
       then
11         replaceNewArrayO
           (pp,M,arg,size);
12         adjustConditionals (M,31);
13     else replaceMultiNewArrayO
       (pp,M,arg,size);
14     adjustConditionals (M,30);
15 else raise error
```

(b) Replacing the `new`, `newarray`, `anewarray` and `amultinewarray` bytecodes

Fig. 13. The back-end code generation pseudo-algorithm.

its `Integer` type reference field `b`. Next, `main` calls method `MCall1`. Field `A` is emitted via signal `S` in method `MCall1`. Our data-flow analysis algorithm needs to locate the program points in `BB1` lines 1 and 2, so that these object allocations can be moved to the permanent heap space.

Part-1 of Step6 calls Step7 passing it method `MCall1` to find out the object `A` and its field's allocation program points. Step7 being an *intra-procedural* analysis step returns back an empty list ($alloc\_pps = \emptyset$). In such a case, Step6 looks up the call-graph tree to find out the caller methods for the current callee, this lookup procedure also includes looking up all the potential callers indicated by the GRC-analysis (c.f. Section 5.1). Once the caller method is identified in the call-graph, Step6 recursively calls itself to analyze the caller. This procedure is continued until the program point allocating the affected field is identified or the analysis reaches the top of the call graph, in the latter case a `Not initialized field` error is thrown by the compiler. In case of Fig. 12, there is a single caller: the `main` method in the call-graph for callee `MCall1`, which is analyzed to find the program point allocating field `A`, this program point is returned by Step6. Note that multiple callers might call the same callee, due to *may* happen before results produced by the GRC-analysis. In such cases, all the callers need to be analyzed in order to identify the program points allocating the affected field and to make sure that such allocations exist in all paths, else, the field would be uninitialized in some run of the SystemJ program.

This finishes the treatment of the data-flow analysis algorithm to identify the program points that return an object allocation reference, which may be emitted via signals. Now that we have identified these program points, we next give the procedure to generate the back-end code that: (1) replaces the object allocation bytecodes with time predictable alternatives and (2) performs compile time memory allocation in the permanent heap space.

6. Back-end code generation

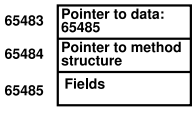
The back-end code generation is a two step-procedure as shown in Fig. 13a and b, respectively. The memory allocation algorithm (Fig. 13a) takes as input the maximum physical memory address, the program points to replace and the whole program itself as input. Two variables: *allocPtr* and *header_size* are initialized to the

```
--- constant-pool snippet ----                          1
#2 = Class      #79     //a/a                           2
#3 = Methodref #2.#78  //a/a.''<init>'':()V            3
#4 = Fieldref  #12.#80 //test/test.A:La/a;             4
---------- main method snippet --------                 5
0:new #2           // class a/a                        6
3:dup                                                   7
4:invokespecial #3 // Method a/a.''<init>'':()V        8
7:putstatic     #4 // Field A:La/a;                     9
```

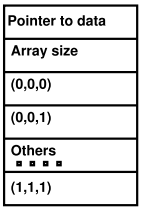
(a) Java bytecode snippet and class constant-pool for example in Figure 12

```
--- constant-pool snippet ----                          1
#2 = Class      #79     //a/a                           2
#3 = Methodref #2.#78  //a/a.''<init>'':()V            3
#4 = Fieldref  #12.#80 //test/test.A:La/a;             4
.... //other constants                                  5
#154 = Integer            65483                         6
#155 = Integer            65485                         7
---------- main method snippet --------                 8
0:ldc_w         #154 // int 65483                       9
3:dup                                                   10
4:ldc_w         #155 // int 65485                       11
7:swap                                                  12
8:invokestatic  #151 // Method Native.wr:(II)V          13
11:dup                                                  14
12:bipush        1                                      15
14:iadd                                                 16
15:ldc_w         #2   // class a/a                      17
18:bipush        5                                      18
20:iadd                                                 19
21:swap                                                 20
22:invokestatic  #151 // Method Native.wr:(II)V         21
25:ldc_w         #155 // start of fields               22
28:ldc_w         #156 // size of the object            23
31:invokestatic  #150 // Method Native.wrl:(II)V        24
34:dup                                                  25
35:invokespecial #3   // Method a/a.''<init>'':()V     26
38:putstatic     #4   // Field A:La/a;                  27
```

(b) Java bytecode and class constant-pool for example in Figure 12 after `new` bytecode replacement



(c) Java object allocation



(d) Java (2x2x2) array allocation

Fig. 14. The back-end generated code and compacted object representation in permanent heap.

maximum memory address and the constant two, respectively. The *allocPtr* is decremented by the size of the object to be allocated (obtained from the data-flow analysis algorithm, Fig. 8) plus, the object header size (line 6) – this simple *bumping* of the allocation pointer is termed bump pointer memory allocation. If the resultant *allocPtr* value is less than 0, then we have exhausted total permanent memory allocation and correspondingly an error is raised (line 8). If there is enough memory available, then the object allocation bytecode at the program point is replaced by alternative bytecodes in the second step, shown in Fig. 13b.

The algorithm (Fig. 13b) used to replace the object allocation bytecodes takes as input four arguments: the allocation pointer: *allocPtr*, the program point to replace: *pp*, the method containing the program point: *M* and the size to allocate: *size*. There are *four* types of bytecodes in the JVM specification: (1) the `new` bytecode that allocates an object. (2) The `newarray` bytecode that allocates arrays holding primitives, (3) `anewarray` bytecode that allocates arrays holding references, and (4) `amultinewarray` bytecode that allocates multidimensional arrays of type identified from the constant pool. This distinction is important since the number of bytes used to represent each of these bytecodes differ. The `amultinewarray` bytecode takes four bytes in the class file. The `new` and `anewarray` bytecodes are encoded in the class file with three bytes, whereas the `newarray` bytecode only uses two bytes in the class file representation.

Different methods are called to replace each of these allocation variants: Fig. 13b, line 3 replacing `new`, line 8 replacing `newarray`, and lines 11 and 13 replacing the `anewarray` and `amultinewarray` bytecodes, respectively. All methods have the same logic, except that the target addresses of jump bytecodes need to be correctly updated after replacement. The target addresses of all branching bytecodes' that follow the object allocation bytecode(s), need to be updated. In case of the `new` bytecode the branching bytecode' target addresses need to be incremented by 31 bytes after replacement.[8] This 31 byte increment is made obvious by Fig. 14a and b, which show the Java bytecode snippets for the example program in Fig. 12 before and after `new` bytecode replace-

[8] In case of `newarray` bytecode the following target addresses are updated by an offset of 32 bytes. In case of `anewarray` the update offset is 31 bytes, and in case of `amultinewarray` it is 30 bytes, respectively

ment, respectively.[9] Fig. 14a shows the bytecode produced by the Java compiler for the `A` object allocation in the `main` method of Fig. 12. The very first bytecode, `new`, takes as argument the index, 2, into the constant pool that holds the type of the object to allocate. The returned object reference is then duplicated using `dup` instruction. Next, the constructor of the class `a` is invoked to initialize the returned reference and, finally the returned reference is put into a static field of the class. The `new` bytecode is timing *unpredictable*, because it might invoke the GC (e.g., in case of work based GCs [7]), which has an unbounded collection cycle. It might also generate a runtime out of memory exception, thereby violating safety criticality. We replace this `new` bytecode with safe and time predictable bytecodes as shown in Fig. 14b.

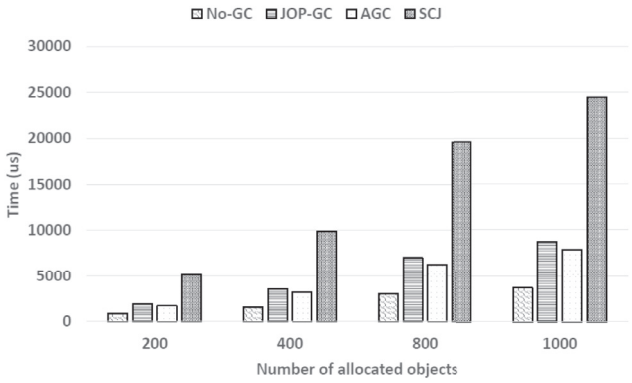
A single `new` bytecode (consuming 3 bytes in the class file) is replaced with a list of bytecodes in lines 9–21 (line numbers are given on the right) consuming 34 bytes in the class file, hence the 31 byte increment of target addresses of the conditional bytecodes. The core idea is very simple; since we have already allocated memory for the object (Fig. 13a), we can simply replace the `new` bytecode with a bytecode that loads the allocation pointer (*allocPtr*) onto the stack (line 9) as the object reference. The rest of the bytecodes setup the object header for the allocated object. Our object header is two words (the object structure for the running example is shown in Fig. 14c), the first word contains the pointer to the start of data (in the current example it is the address 65485). The first word of the object header is initialized in lines 11–13 in Fig. 14b. The second word of the object header is the pointer to the start of the method structure of the class (c.f. Fig. 5b). The second word of the object header is initialized in lines 14-21 in Fig. 14b. Finally, the fields of the object are zeroed, on lines 22–24. The method `Native.wrl` takes as input the start address of the object's fields and the size of the object as the arguments and writes zeros in these memory words using a loop.

In case of `newarray`, `anewarray`, and `amultinewarray` bytecodes, the array layout (shown in Fig. 14d) differs from standard real-time JVM implementations. For array allocation, we again use a 2 word object header. The first word contains the pointer to the start of array data. The second word of the object header contains the size of the array. Other than the smaller object header size compared to standard real-time JVM object header size, which uses spines [27] or handles [28] for arrays and regular objects, respectively, we also structure our arrays differently. We have decided to place arrays in a row major order and contiguously (as in 'C') in order to allow $O(1)$ array accesses.

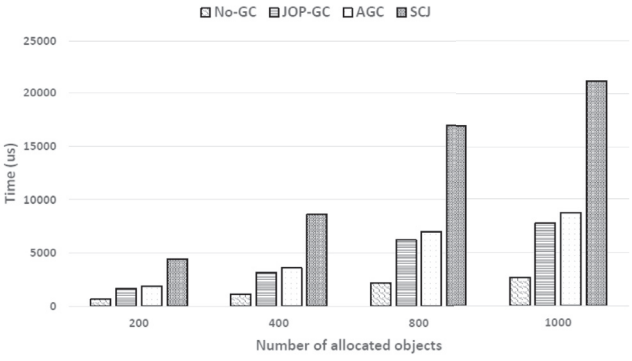
7. Experimental results

We have carried out two sets of experiments: (a) micro-benchmarks and (b) static WCRT analysis on well established real-time benchmarks to study the efficacy of the proposed memory management scheme compared to: (1) two different real-time garbage collector (RTGC) implementations [28,29] and (2) The *Safety Critical Java* (SCJ) [30] implementation on the JOP platform. The first RTGC framework is the one proposed in [28], which is a concurrent version of Cheney's copy collector developed by Baker [31]. In the rest of the section we call it JOP-GC. The second one proposed in [29] is a variation of the real-time collector described in [27], in the rest of the section we call it AGC. The SCJ framework on the other hand is a GC less Java specification, designed to build hard real-time safety critical applications. All our micro-benchmark experiments are run in ModelSim, performing cycle-accurate simulation of the real-time hardware implemented JVM called JOP, running at 100 MHz, with 256KB of main memory, 1KB of method cache, and 4KB of stack cache.

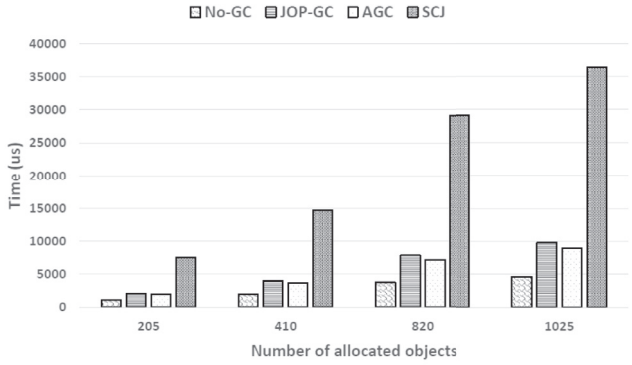
[9] In both these figures, the line numbers are shown on the right of the program. The numbers adjacent to the bytecodes are the starting byte offsets for each of the bytecodes as obtained by the `javap -c -v` command.



(a) Simple object memory allocation runtime without GC invocation.



(b) Simple array memory allocation runtime without GC invocation.

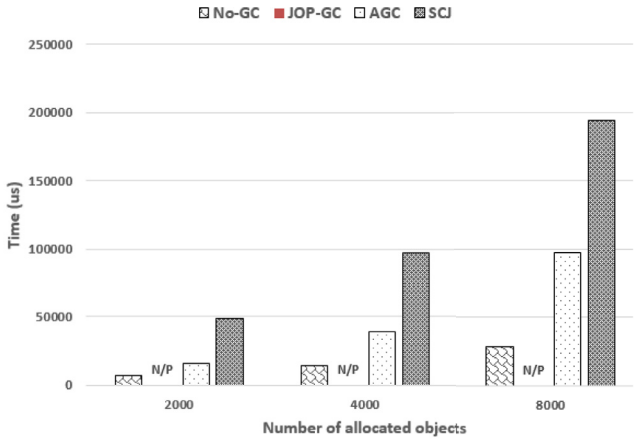


(c) Complex object memory allocation runtime without GC invocation.

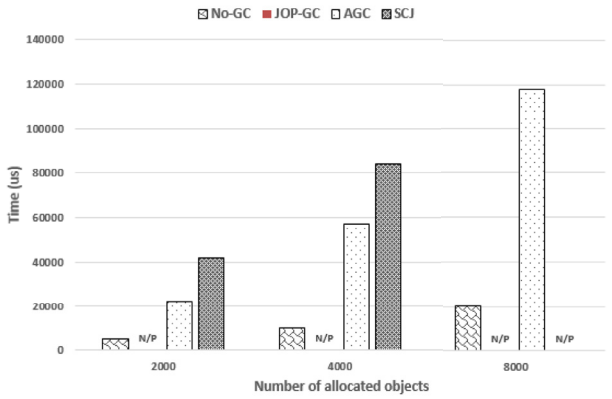
Fig. 15. Runtime for micro-benchmarks without GC invocation.

7.1. Micro benchmarks

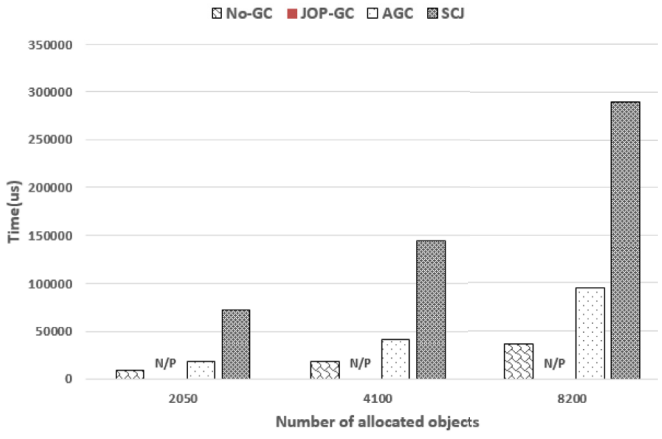
We experiment with three micro-benchmarks: (1) simple object allocation, where we allocate, in a tight loop, 2 objects with 1 word primitive field each. (2) Simple array allocation, where we allocate, in a tight loop, an object with an array type reference field, which is itself initialized to a 20 word 1D array of primitive `int` type. (3) Complex object allocation: in this case, we use a nested 2D loop. In every outer loop iteration, a *reference* type 1D array of size 20 is allocated. In every iteration of the inner loop, 2 objects



(a) Simple object memory allocation runtime with possible GC. **N/P** stands for `NullPointer` exception



(b) Simple array memory allocation runtime with possible GC invocation. **N/P** stands for `NullPointer` exception



(c) Complex object memory allocation runtime with possible GC invocation. **N/P** stands for `NullPointer` exception

Fig. 16. Runtime for micro-benchmarks with possibility of GC invocation.

are allocated and the second object's reference is set as the array value. The example benchmarks are available from Pathirana [29].

The exact same micro-benchmarks, all written in Java, are executed in all four approaches. Only the back-end memory management technique differs in the four executions. The runtime comparison between the four approaches for each of the micro-benchmarks is shown in Figs. 15 and 16. Figs. 15 and 16 give the execution times for completion of a memory allocation request in two cases: (1) when the GC is not invoked, i.e., there is enough memory available to allocate the requested memory and (2) when the GC is invoked to collect the garbage when enough memory is not available upon a memory allocation request from the mutator, respectively. The proposed approach is, on average, approximately $3\times$ faster compared to the RTGC based approaches (Table 1). In many cases (see Fig. 16) JOP-GC could not perform garbage collection correctly, as it ran out of handles resulting in `NullPointer` exceptions. There is a greater speedup in case of array allocations compared to simple object allocations. The runtime memory allocation time difference is significant in the case of GC invocation when compared to the case of no GC invocation.

SCJ specification prohibits GC [30] and divides the memory into regions such as immortal memory and scoped memory, similar to the permanent and transient heap spaces described in this paper. A more thorough comparison between the proposed approach and the SCJ memory organization is given later in Section 8.1. Given this similarity in the memory organization schemes, we expected to observe memory allocation times, for SCJ, similar to those obtained in the No-GC approach proposed in this paper. Yet, as we see from Figs. 15 and 16, the SCJ memory allocation times are far larger compared to all the other memory allocation approaches. In particular, SCJ's memory allocation implementation, in our experiments, is, on average, $\approx 7.3\times$ slower than the proposed approach. Close analysis of the SCJ implementation on the JOP platform indicates that the memory allocation code consists of a number of branch statements, which cause this slowdown. We expect that a better memory allocation implementation would result in faster memory allocation times for SCJ, too.

In case of memory allocation time when no GC is invoked, there are multiple reasons for the speedup in the No-GC case, compared to other approaches: (1) there are no read/write barriers when allocating memory, which are needed by JOP-GC, AGC, and SCJ. (2) The memory allocation bytecodes are implemented in software, which means, on *every* execution of these bytecodes, a method may be loaded into the method cache, which results in increased program latency. (3) In case of the array allocation bytecode execution, array spines need to be initialized for AGC [29,32], which leads to further slow-down in memory allocation times.

Table 2 gives the number of words allocated for the complex object allocation benchmark for all the four approaches. In the proposed approach, every iteration of the loop reuses the space allocated on the permanent heap space. The RTGC approaches on the other hand, allocate memory on each allocation bytecode execution. Our compile time memory allocation approach simply replaces these memory allocation bytecodes as described in Section 6 and hence, we do not keep on allocating more words. On the other hand, the RTGC approaches, inherently allocate memory on every object and array allocation call. Another important point to note is that AGC although faster when it comes to memory allocation runtime compared to JOP-GC, allocates more words, because AGC uses block based allocation, which leads to internal memory fragmentation, whereas JOP-GC uses object replication strategy with semi-space partitioning.[10]

7.2. WCRT comparisons

In this section we compare the static WCRT obtained by applying the technique described in Section 3.2 on a set of real-time benchmarks. We chose the benchmarks whose memory allocation requests fit within the allocated heap space without GC invocation. Consequently, we also manually-deleted all code related to

[10] In the numbers in Table 2, we have not included the semi-space reserved by JOP-GC.

Table 1
Garbage collection methods compared in experimental results.

Method	Description	Status of work
No-GC	SystemJ program compiled to Java without GC	Proposed work
JOP-GC	Java programs executing on a Java Optimized Processor with software GC	Existing work [28]
AGC	Java programs executing on JoP with a variation of fragmentation-tolerant real-time garbage collection	Existing work [29]
SCJ	Safety Critical Java Implementation without GC	Existing work [30]

Table 2
Comparison of memory words allocated for the complex object allocation. Each word is 4 bytes.

# of loop iterations	No-GC (Words)	JOP-GC (Words)	AGC (Words)	SCJ (Words)
100	28	2145	2995	2145
200	28	4290	5590	4290
400	28	8580	11180	8580
500	28	10,725	13,975	10,725
1000	28	21,450	29,950	21,450
2000	28	42,900	55,900	42,900
4000	28	85,800	111,800	85800

GC. This included deleting the complete garbage collection method call itself. This was a necessary step, because one cannot *practically* bound the GC cycle time for purpose of static analysis. Puffitsch [6] have already shown that GC cycle times are orders of magnitudes larger than the execution times of real-time tasks. Explicitly deleting all GC related code also makes it a fair comparison. The JOP-GC WCRT results are not shown, because JOP-GC always results in `NullPointer` exceptions for all these examples. We also do not compare with SCJ because: (1) the open source JOP based SCJ implementation is naïve, resulting in large slowdowns during memory allocations, as seen in Figs. 15 and 16 and (2) all the real-time benchmark programs are reactive, and run in an infinite loop allocating and releasing memory. SCJ specification does not allow reuse of memory words, in the scoped memory area, until the real-time task completes execution (as seen in Table 2) and hence, all our real-time benchmarks exhaust all memory when executed as SCJ tasks.

We chose 5 real-time synchronous programs for bench marking. `Cruise control` [33] is the cruise speed controller found in cars. `Hrtcs` [34] is a human response time gathering system, `Conveyor controller` is a fruit sorter controller, which controls placement of fruits on a conveyor belt using image processing a more detailed description of the application is provided in [14]. `Motor` [35] is a stepper motor control system for a printer and finally, `Pacemaker` [13], is a real-time pacemaker used to control arrhythmia. These benchmarks are compiled to the TP-JOP architecture for execution and their WCRT values are analyzed. All benchmarks are written in SystemJ, and only the backend memory management techniques differ for the two approaches being compared. The benchmarks are available from WCRT [36]. The results are shown in Fig. 17a.

Ignoring the `Motor` example, on average, the proposed No-GC approach's WCRT is around 23% shorter compared to AGC. The `Motor` example is an outlier, with a 4000% improvement in WCRT, because a number of object allocations are performed when emitting events to control the motor coils. In other examples, object allocations are used in computations on the transient heap space, with relatively fewer object allocations on the permanent heap. This requires deep copying objects from the transient to the permanent heap space, which balances out the WCRT times between the No-GC and the AGC approaches. Note that, the method cache load times and synchronized heap access overheads are still present in the AGC approach.

Finally, Fig. 17b gives the average method size comparisons for the compiled SystemJ programs. Since we replace a single object allocation bytecode with a list of other bytecodes, the proposed approach increases the method size. Consequently this also has a penalty on the computed WCRT value, in case of the proposed approach, as the method cache load time increases, but it is not as significant as loading the methods implementing the object allocation for every memory allocation bytecode execution in the AGC approach.

8. Related work and discussion

8.1. Comparison with region based, SCJ and RTSJ memory management approaches

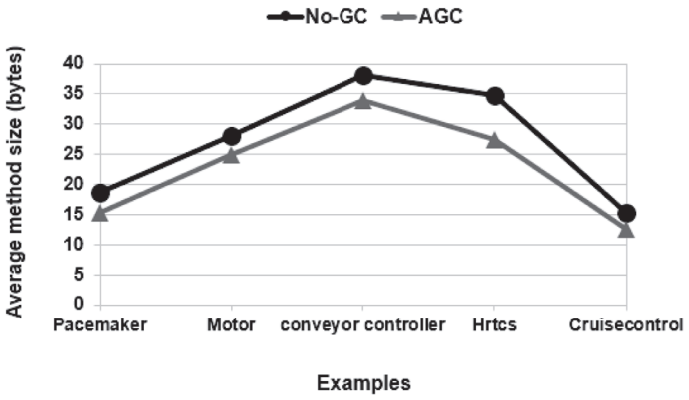
We are not the first to advocate a GC-less approach for hard real-time applications with memory managed languages, especially Java. Earlier works on *region* based memory management [37,38] advocate the use of different regions, similar to transient and permanent heap proposed in this work, for safely managing memory. The work of Nguyen and Rinard [38] uses cyclic memory allocation/deallocation scheme per allocation site in C programs. They empirically estimate the size and the number of memory allocations for each allocation site in the program using trial runs. Once both these quantities are estimated, a $N \times S$ memory region is dedicated for allocation for each allocation site, where N and S are the number of allocations and size of objects to be allocated for that site obtained from the trial runs. The problem with this approach is that the trial runs are not exhaustive, i.e., the number of allocations for any site can potentially be underestimated. Such underestimation can lead to overwriting live objects, resulting in wrong program execution.

The work in [37] proposes a region based memory management approach for C. In their approach, a reference count for complete regions is used for garbage collection. Managing reference counts results in additional overhead during program execution. Gay and Aiken in [37] posit that region based memory management is useful only when life-time of objects can be determined statically. SystemJ's MoC allows exactly determining the life-time of objects, due to the state demarcation approach inherent in synchronous programming. Hence, our proposed approach does not incur any overheads (like reference counting) and leverages the synchronous MoC to provide an elegant region based memory management solution unlike any previous approaches.

There also exists previous works like the Safety Critical Java (SCJ) [30] specification, Ravenscar-Java [39] and Real Time Specification for Java (RTSJ) [40] that present a memory organization ap-

Benchmarks	No-GC	AGC
Pacemaker	14424205	19485761
Motor	2674	1087387
Conveyor controller	3325301	4048418
Hrtcs	318	318
Cruise control	2218493	2941605

(a) WCRT values (in clock cycles) for benchmarks without GC invocation



(b) Method sizes for benchmarks

Fig. 17. WCRT and method size comparison: No-GC vs. AGC.

```
A a = null;                                    1
for (int i = 0; i< N; ++i)                     2
 a = new A();                                  3
emit S(a);                                     4
```

Fig. 18. SystemJ code snippet to show reuse of memory words.

proach similar to the one presented in this paper. SCJ, Ravenscar-Java and the RTSJ approaches divide the backing store into multiple parts. A memory region called scoped memory is allocated per real-time task (called a schedulable object in SCJ/RTSJ terminology), which is automatically reclaimed, using pointer reset like in our case, once the task completes execution. There also exists a permanent memory area, called the immortal memory, similar to the permanent heap advocated in this paper. Any object allocated in the immortal memory lives throughout the lifetime of the application. One major difference between the presented memory partitioning approach and the memory partitioning in SCJ and Ravenscar-Java approaches is that in the presented approach the same transient heap space (similar to scoped memory in SCJ) can be shared by multiple synchronous parallel reactions, whereas every single task gets allocated its own scoped memory (which might be nested in another tasks' scoped memory) in SCJ. This is because, SCJ allows context switching of tasks during program execution, whereas in SystemJ a clock-domain tick is atomic. RTSJ does allow access of scoped memory by concurrent threads and only when the last thread exits, the scoped memory is reclaimed. Another important point of differentiation is the reuse of memory words enabled by the proposed approach. Consider the SystemJ code snippet in Fig. 18. An object `a`, of type `A` is initialized multiple times on lines 2–3. The proposed approach *replaces* the `new` bytecode with simple load instructions, hence, the object does not get allocated multiple times in the loop. In fact, the same memory location (address/words) is reused by the proposed approach. In the SCJ/RTSJ memory organization, every memory allocation bytecode (e.g., `new`) allocates new memory words and hence, reuse is impossible.

Furthermore, in the approach proposed in this paper the compiler automatically decides where to place the allocated objects, whereas in the SCJ and RTSJ approaches, the programmer needs to explicitly and carefully allocate objects to the correct memory areas manually. We believe that our approach is safer and helps speed up development times.

The proposed algorithms also have a lot in common with a number of tools proposed for easing development of programs in SCJ and RTSJ. The worst case memory analyzer tool [9] developed for SCJ is able to provide the maximum permanent memory and scoped memory sizes needed for each task automatically, like in our case. But, the programmer is still responsible for reserving the appropriate sized permanent memory and scoped memory areas in the backing store. Compiler driven memory allocation is a non-trivial task for SCJ and RTSJ applications, because SCJ and RTSJ allow free use of the Java language, consequently bringing all the disadvantages such as: pointer aliasing, shared memory communication, etc, which we restrict in the SystemJ MoC. Finally, JVMs other than JOP can be used to execute SystemJ programs with the proposed memory organization, provided an array based backing store is available.

8.2. Comparison with real-time garbage collection approaches

Real-time garbage collectors (RTGCs) such as the ones proposed in [41,42] and the ones presented in [32], aim to achieve good garbage collection throughput while providing analytic solutions to the worst case execution times of a *single* GC cycle. Once the GC cycle times are obtained there is the need to schedule the GC work. The scheduling can be either *work* based where the GC work is performed at object allocation by the already present real-time tasks, or *time* based where the GC runs as its own task and needs to be scheduled with the other real-time tasks in the system. Both these approaches suffer from drawbacks, such as determining the priority of the GC task. The major drawback of these approaches is that the equations capturing the GC cycle times [7] depend upon the memory allocation characteristic of the application. In the general case, assuming the worst case memory allocation patterns results in a very pessimistic GC worst case bound as shown in [6]. In this paper we are trying to remedy this problem by removing the GC altogether. More recently, a hardware reference counting unit, for hard real-time Java applications, has been proposed in [43], which overcomes the aforementioned RTGC drawbacks. In the approach of Harvey-Lees-Green et al. [43], a hardware reference counting unit runs in parallel with the processor core, counting references and collecting objects when the reference counts for the objects becomes zero. The proposed work in contrast to Harvey-Lees-Green et al. [43] does not need additional hardware unit for garbage collection. Instead we leverage the SystemJ GALS MoC to remove the GC altogether. Hence, overall we can achieve the same hard real-time guarantees as [43], but *without* additional execution time overheads.

```
List<Integer> list = new List<Integer>();          1
while (true) {                                     2
 // I is a valued input signal                     3
 // from the environment                           4
 present(I) {                                      5
  list.add(new Integer(#I));                       6
  // O is a valued output signal                   7
  emit O(list);                                    8
  pause;                                           9
  list = #O;                                       10
}                                                  11
```

Fig. 19. Example SystemJ code snippet that cannot be compiled using the proposed approach.

8.3. Perceived limitation of the proposed approach

The primary limitation of the approach proposed in this paper is the fact that the compiler should be able to determine the size of the object to allocate statically. Consider the SystemJ code snippet in Fig. 19. In this example, a `List` type object (`list`) is allocated on line 1. New `Integer` type objects are allocated and added to this `list` if and only if the valued signal `I` is present from the environment. Moreover, the value carried by the signal `I` is added into the `list`. In the first tick, the list object is emitted back to the environment via signal `O` (line 8). Finally, the `list` object has a lifetime greater than one tick, because it is reused in the second tick (line 10). Hence, the `list` object has to be allocated into the permanent heap space. We do not know how often signal `I` will be input from the environment and with what value. Thus, we cannot determine at compile time the worst case size of the `list` object. Therefore, in this example, the permanent heap size cannot be bounded at compile time. The proposed approach terminates compilation in such cases stating that the heap size is unbounded.

In all safety critical and hard real-time application, including those developed in SCJ [9], one needs to statically determine the worst case memory consumption. This is important, because runtime "out of memory" exceptions can lead to catastrophic damages. Hence, the need of our approach to statically bound the memory consumption is actually a blessing in disguise.

9. Conclusion and future work

In this paper we have presented a new memory organization and model for hard real-time and safety-critical applications programmed in formal synchronous languages with data support being provided by managed language runtimes, primarily Java. The SystemJ model of computation plays a central role in the presented memory management approach. The "tick" demarcation inherent in the formal synchronous programming model, provided by SystemJ, allows dividing the Java objects into two types: (1) transient objects, which are allocated for a single clock-domain transition *only* and are collected by simple pointer reset and (2) permanent objects, which are alive throughout the life time of the application. Adhering to the synchronous model of computation, allows us to perform compile time memory allocation, which means that our programs are guaranteed to be free of execution time *out of memory* exceptions. Furthermore, we are able to replace object allocation bytecodes, with real-time analyzable alternatives, which makes our approach amenable to non-pessimistic worst case execution time analysis. Java lacks a clear state ("tick") demarcation and, hence, dividing memory, and associated objects, into efficient collectable memory spaces is a challenge.

All the aforementioned qualitative improvements are further accompanied by improvements in the program throughput (approximately 3× faster memory allocation times) and worst case execution times, as synchronization barriers necessary in real-time garbage collection approaches become unnecessary in the proposed approach. We also demonstrate the improvements in the number, size and compacted layout of allocated Java objects and arrays.

We see opportunities for future optimization, especially relaxing the object escapement restrictions currently enforced by the proposed data-flow analysis. We also see an opportunity to *reuse* permanent heap memory allocations across clock-domain tick boundaries, which we plan to address in the future.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.micpro.2018.10.007.

References

[1] G. Berry, G. Gonthier, The Esterel synchronous programming language: design, semantics and implementation, Sci. Comput. Program. 19 (2) (1992) 87–152.

[2] L.J. Jagadeesan, C. Puchol, J.E. Von Olnhausen, Safety property verification of Esterel programs and applications to telecommunications software, in: Computer Aided Verification, Springer, 1995, pp. 127–140.

[3] A. Malik, Z. Salcic, P.S. Roop, A. Girault, SystemJ: a GALS language for system level design, Els. J. Comput. Lang. Syst. Struct. 36 (4) (2010) 317–344.

[4] M. Boldt, C. Traulsen, R. von Hanxleden, Worst case reaction time analysis of concurrent reactive programs, Electron Notes Theor. Comput. Sci. 203 (4) (2008) 65–79.

[5] P.S. Roop, S. Andalam, R. von Hanxleden, S. Yuan, C. Traulsen, Tight WCRT analysis of synchronous C programs, in: International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, 2009, pp. 205–214.

[6] W. Puffitsch, Design and analysis of a hard real-time garbage collector for a Java chip multi-processor, Concurr. Comput. 25 (16) (2013) 2269–2289.

[7] M. Schoeberl, Scheduling of hard real-time garbage collection, Real-Time Syst. 45 (3) (2010) 176–213.

[8] M. Schoeberl, R. Pedersen, Wcet analysis for a java processor, in: Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006), ACM Press, 2006, pp. 202–211. https://doi.org/10.1145/1167999.1168033.

[9] J.L. Andersen, M. Todberg, A.E. Dalsgaard, R.R. Hansen, Worst-case memory consumption analysis for SCJ, in: Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems, ACM, 2013, pp. 2–10.

[10] C. Hoare, Communicating sequential processes, Commun. ACM 21 (8) (1978) 666–677.

[11] Z. Salcic, A. Malik, GALS-HMP: a heterogeneous multiprocessor for embedded applications, ACM Trans. Embedded Comput. Syst. 12 (1s) (2013) 58, doi:10.1145/2435227.2435254.

[12] M. Nadeem, M. Biglari-Abhari, Z. Salcic, RJOP: a customized Java processor for reactive embedded systems, in: Proceedings of the 48th Design Automation Conference, ACM, 2011, pp. 1038–1043.

[13] H. Park, Avinash Malik, M. Nadeem, Z.A. Salcic, The Cardiac Pacemaker: SystemJ versus Safety Critical Java, in: Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES 2014, Niagara Falls, NY, USA, October 13–14, 2014, 2014, p. 37, doi:10.1145/2661020.2661030.

[14] Z. Li, Avinash Malik, Z.A. Salcic, TACO: a scalable framework for timing analysis and code optimization of synchronous programs, in: 2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications, Chongqing, China, August 20–22, 2014, 2014, pp. 1–8, doi:10.1109/RTCSA.2014.6910556.

[15] M. Schoberl, JOP: a Java optimized processor, Workshop on Java Technologies for Real-Time and Embedded Systems, 2003.

[16] M. Schoeberl, Design and implementation of an efficient stack machine, in: Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005), IEEE, 2005. https://doi.org/10.1109/IPDPS.2005.161

[17] M. Schoeberl, A time predictable instruction cache for a java processor, in: On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004), vol. 3292, Springer, 2004, pp. 371–382. https://doi.org/10.1007/b102133.

[18] B. Becker, A. Lastovetsky, Max-plus algebra and discrete event simulation on parallel hierarchical heterogeneous platforms, Euro-Par 2010/HeteroPar 2010, Ischia-Naples, Italy, 2010.
[19] P.S. Roop, S. Andalam, R. von Hanxleden, S. Yuan, C. Traulsen, Tight WCRT Analysis of Synchronous C Programs, Technical Report 0912, Christian-Albrechts-Universität Kiel, Department of Computer Science, 2009. Www.ece.auckland.ac.nz/~roop/pub/2009/roop-report09.pdf
[20] S. Andalam, P.S. Roop, A. Girault, Pruning infeasible paths for tight WCRT analysis of synchronous programs, in: Design, Automation and Test in Europe, DATE 2011, Grenoble, France, March 14–18, 2011, 2011, pp. 204–209.
[21] H. Theiling, C. Ferdinand, R. Wilhelm, Fast and precise wcet prediction by separated cache and path analyses, Real-Time Syst. 18 (2–3) (2000) 157–179.
[22] G. Behrmann, A. David, K.G. Larsen, A tutorial on uppaal, in: M. Bernardo, F. Corradini (Eds.), International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures, Lecture Notes in Computer Science, vol. 3185, Springer Verlag, 2004, pp. 200–237.
[23] S.S. Muchnick, Advanced Compiler Design Implementation, Morgan Kaufmann, 1997.
[24] J.-D. Choi, M. Gupta, M. Serrano, V.C. Sreedhar, S. Midkiff, Escape analysis for Java, in: ACM Sigplan Notices, 34, 1999, pp. 1–19.
[25] J. Meyer, T. Downing, Java Virtual Machine, O'Reilly & Associates, Inc., 1997.
[26] M. Burke, P. Carini, J.-D. Choi, M. Hind, Flow-insensitive interprocedural alias analysis in the presence of pointers, in: Languages and Compilers for Parallel Computing, Springer, 1995, pp. 234–250.
[27] F. Pizlo, L. Ziarek, P. Maj, A.L. Hosking, E. Blanton, J. Vitek, Schism: fragmentation-tolerant real-time garbage collection, in: ACM Sigplan Notices, 45, ACM, 2010, pp. 146–159.
[28] M. Schoeberl, JOP: A Java Optimized Processor for Embedded Real-Time Systems, Ph.D. thesis, Vienna University of Technology, 2005.
[29] I. Pathirana, AucklandGC: A Real-Time Garbage Collector for the Java Optimized Processor, Master's thesis, University of Auckland, 2013.
[30] JSR 302: Safety Critical Java Technology, 2013. (http://jcp.org/en/jsr/detail?id=302)
[31] H.G. Baker Jr, List processing in real time on a serial computer, Commun. ACM 21 (4) (1978) 280–294.
[32] F. Pizlo, E. Petrank, B. Steensgaard, A study of concurrent real-time garbage collectors, in: ACM SIGPLAN Notices, 43(6), ACM, 2008, pp. 33–44.
[33] A. Charles, Representation and analysis of reactive behaviors: a synchronous approach, in: Computational Engineering in Systems Applications, CESA, vol. 96, 1996, pp. 19–29.
[34] H. Park, A. Malik, Z. Salcic, Time Square – marriage of real-time and logical-time in GALS and synchronous languages, in: 2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2014, 2014.
[35] T. Bourke12, A. Sowmya, Delays in esterel, in: SYNCHRON09, 2009, p. 55.
[36] WCRT Benchmarks, (https://bitbucket.org/amal029/sysjexamples/src). Last accessed - 22.08.2017.
[37] D. Gay, A. Aiken, Memory management with explicit regions, ACM SIGPLAN Notices, 33(5), ACM, 1998.
[38] H.H. Nguyen, M. Rinard, Detecting and eliminating memory leaks using cyclic memory allocation, in: Proceedings of the 6th international symposium on Memory management, ACM, 2007, pp. 15–30.
[39] J. Kwon, A. Wellings, S. King, Ravenscar-Java: a high-integrity profile for real–time Java, Concurr. Comput. 17 (5–6) (2005) 681–713.
[40] G. Bollella, J. Gosling, The real-time specification for java, Computer 33 (6) (2000) 47–54.
[41] S. Gestegard-Robertz, R. Henriksson, Time-triggered garbage collection, in: Proceedings of the ACM SIGPLAN Langauges, Compilers, and Tools for Embedded Systems, 2003.
[42] T. Kalibera, F. Pizlo, A.L. Hosking, J. Vitek, Scheduling hard real-time garbage collection, in: Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE, IEEE, 2009, pp. 81–92.
[43] N. Harvey-Lees-Green, M. Biglari-Abhari, A. Malik, Z. Salcic, A dynamic memory management unit for real time systems, in: 2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC), 2017, pp. 84–91, doi:10.1109/ISORC.2017.4.

Avinash Malik is a lecturer at the University of Auckland, New Zealand. His main research interest lies in programming languages for multicore and distributed systems and their formal semantics/ compilation. He has worked at organizations such as INRIA in France, Trinity College Dublin, IBM research Ireland, and IBM Watson on design and compilation of programming languages. He holds B.E. and Ph.D. degrees from the University of Auckland.

Heejong Park received his B.E. degree in computer systems engineering and Ph.D. degree in electrical and electronics engineering in 2010 and 2016, respectively, both from the University of Auckland, Auckland, New Zealand. He is currently working as a postdoctoral research fellow in the Department of Electrical and Computer Engineering, the University of Auckland. His main research interests include compilation and verification techniques for safety-critical systems, formal semantics of programming languages, and real-time computing.

Muhammad Nadeem received his B.Sc. engineering degree in electrical and electronics from University of Engineering and Technology, Lahore, Pakistan, in 1998. He received his M.Sc. engineering degree from Royal Institute of Technology, Sweden, and Ph.D. degree from The University of Auckland, Auckland, New Zealand. He is currently a Professional Teaching Fellow with the Department of Electrical and Computer Engineering, The University of Auckland, Auckland, New Zealand. His current research interest include embedded system, Java processors, multiprocessors systems, real-time systems and mixed-criticality systems.

Zoran Salcic holds a chair in computer systems engineering at the University of Auckland. He has the BE ('72), ME ('74) and PhD ('76) degrees in electrical and computer engineering (Sarajevo University). His main research interests include complex digital systems, custom-computing machines, embedded systems and their implementation, design automation tools, hardware software co-design, models of computation and languages for concurrent and distributed systems and cyber-physical systems. He has published more than 300 peer-reviewed journal and conference papers and several books. He is a Fellow of the Royal Society New Zealand and recipient of Alexander von Humboldt Research Award in 2010.