Compiler assisted memory management for safety-critical hard-real time applications

Avinash Malik, University of Auckland
HeeJong Park, University of Auckland
Muhammad Nadeem, University of Auckland
Zoran Salcic, University of Auckland

Safety critical hard real-time applications need to be statically analyzable for guaranteeing functionally correct operation and bounding their worst case execution time (WCET). Applications programmed in managed languages such as Java depend upon garbage collection (GC) for memory management. GC cycle times cannot be realistically bounded, since they depend upon the memory allocation patterns of the application. Pessimistic WCET bounds of applications developed in managed languages are orders of magnitude larger than their real execution times and hence, impractical for real-world development. Real-time GC (RTGC) approaches also require task preemption, which leads to intractable state space explosion during automated formal verification. In this paper we propose a programming model driven memory organization, allocation, and garbage collection approach that delivers non-pessimistic WCET bounds for applications developed in a managed language. Furthermore, we base our work on programming languages based on formal mathematical semantics, thereby allowing for formal reasoning of the developed applications.

We develop a compiler assisted memory management technique for safety critical hard real-time applications developed in the synchronous subset of the formal globally asynchronous locally synchronous programming language called SystemJ. The SystemJ model of computation allows us to partition the heap into two distinct areas and perform compile time memory allocation. The new memory reclaim procedure is a simple pointer reset, which is guaranteed to complete within a bounded number of clock-cycles, thereby alleviating the need for pessimistic WCET bounds. Other than being amenable to formal verification and tight WCET analysis, results show that our proposed approach to memory management is approximately $3\times$ faster compared to standard RTGC approaches.

Categories and Subject Descriptors: D.3.4 [**Processors**]: Compilers

General Terms: Compiler, Static Analysis, WCET, SystemJ, Garbage Collection

Additional Key Words and Phrases: Wireless sensor networks, media access control, multi-channel, radio interference, time synchronization

ACM Reference Format:
ACM Trans. Embedd. Comput. Syst. , , Article (November 2014), 25 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION AND RELATED WORK

With the advent of the Internet of Things (IoT) paradigm, the line between embedded real-time computing and standard desktop computing is blurring. As more sensors and actuators are connected to the embedded devices, the software code bases controlling these devices is only predicted to grow in complexity. In order to manage this inherent complexity, managed programming languages such as Java are being considered even

Author's addresses: Department of Electrical and Computer Engineering, University of Auckland, New Zealand

for safety-critical and hard real-time systems [scj 2013]. There are two main requirements that need to be satisfied when considering real-time and safety-critical systems: (1) respecting the environment specified space and time bounds and (2) guaranteeing functional correctness. Both these aspects have been studied in the research literature quite extensively for managed languages, especially Java [Havelund and Pressburger 2000; Pizlo et al. 2008; scj 2013; Puffitsch 2013].

For any given real-time system, *Worst Case Execution Time* (WCET) of a task should be statically known so that real-time tasks can be scheduled to meet their individual deadlines [Wilhelm et al. 2008]. Computing WCET of a task programmed in a non-managed language such as ‘C’ is a non-trivial task since detailed knowledge of hardware and interaction of the task and the underlying hardware needs to be known a-priori. Managed languages include a runtime environment with *Just In Time* (JIT) compilation, which exacerbates the problem of computing the WCET of the task programmed in a managed language to such an extent as to render it unfeasible. In order to skirt this problem, researches (both academic and commercial) either perform ahead of time compilation of their Java programs [Kalibera et al. 2011; Pizlo et al. 2010] or use hardware implemented Java virtual machines [Schoeberl 2005]. In either case, the JIT compilation interference is no longer a problem in determining the WCET of the task. Instead, the main challenge now is to reconcile the memory allocation and garbage collection phases inherent to a managed language within the WCET static analysis framework.

Many approaches to *Real-time Garbage Collection* (RTGC) have been proposed, a good survey of RTGCs can be found in [Pizlo et al. 2008]. There are multiple requirements that a RTGC needs to satisfy. From the perspective of the real-time task, the main requirement is that: (1) the maximum blocking time for a GC should be bounded and (2) the number of *times* a task may be interrupted by the GC should be bounded. From the perspective of the GC itself, one needs to statically guarantee that enough memory is available to keep pace with the memory allocations demanded by the real-time task.

There are two principle approaches to integrating RTGCs within a real-time framework. The first approach pioneered by Henriksson [Henriksson 1998] requires one to schedule the GC real-time task as the lowest priority thread in slack time – time between the completion of a task and its deadline to *incrementally* collect garbage. The second approach is to use a periodically running GC as the highest priority task that preempts a real-time task and runs for a statically determined amount of time [Bacon et al. 2003] again incrementally collecting garbage. Both approaches require a preemptable real-time system. In the first approach the GC task might be preempted by a higher priority real-time task or an asynchronous event. In the second approach real-time tasks themselves need to be preempted by the GC task. This need for preemptability has two drastic consequences: (1) every access to heap allocated object needs to be guarded by read and write barriers, thereby increasing time for every heap access and also memory allocation itself, which in turn increases the task’s WCET. (2) Automated functional verification of priority preemptive multi-tasking model via approaches such as model-checking [Clarke et al. 2000; Havelund and Pressburger 2000] is known to be intractable due to the exponential state space explosion problem. Thereby making it unfeasible to automatically verify such systems for functional correctness.

The question then becomes: *is programming real-time safety-critical systems using a manged language just impractical?* In this paper we argue that a well thought out programming model along with compiler assisted memory management is not only a practical, but also an efficient approach to programming real-time and safety-critical systems using managed languages.

The most important requirement is that the programming model of the managed language be formally verifiable or at least not be proactively hostile to formal verification. In our opinion a system that meets all its real-time deadlines, but is functionally incorrect is of little use. Hence, we prioritize functional correctness over real-time guarantees, although at the end, both requirements need to be satisfied. In order to adhere to this requirement we consider languages (within the real-time community) based on formal semantics that can be used for automated formal verification. Synchronous languages such as Esterel [Berry 1993], Lustre [Halbwachs et al. 1991], Signal [Le Guernic et al. 1991] and *Globally Asynchronous Locally Synchronous* (GALS) languages such as SystemJ [Malik et al. 2010] have been used to design large real-time and safety-critical systems [Bouali 1998; Park et al. 2014a; Li et al. 2014; Malik et al. 2012]. All these languages are not only based on formal mathematical semantics, but are also designed to be amenable to automated formal verification. Especially imperative languages Esterel and its derivation SystemJ 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 *reacts instantaneously* in zero time to respond to these inputs and becomes quiescent until the next input events arrive. The start and end of this reaction transition demarcate states of the program – note that internal data updates do not lead to change 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 sometime, all one needs to do is find out the *Worst Case Reaction Time* (WCRT) of any given synchronous program. This WCRT value determines the shortest inter-arrival time for input events from the environment.

Many techniques exist for computing WCRT values for synchronous programs providing support for data-computation via a non-managed language (primarily ‘C’). A good review is available in [Wilhelm et al. 2008]. We are interested in the WCRT analysis of synchronous programs that support data computation via managed-languages like Java. The SystemJ [Malik et al. 2010] language provides the synchronous programming model as a subset along with Java driven data-computations, thereby mixing synchronous *Model of Computation* (MoC) with a managed-language runtime environment. WCRT computation techniques have been developed for SystemJ [Li et al. 2014], but the authors do not state anything about the integration of the GC within this WCRT framework.

The main **contribution** of this paper is to present a new memory management approach that is amicable to static WCRT analysis of synchronous programs intertwined with managed runtime environments for data-computation support. More concretely our contributions can be refined as follows:

— *Programming model inspired memory organization*: In this paper we present a new memory organization for Java and accompanying static WCRT analysis based on the SystemJ MoC.
— *Compiler supported memory allocation*: We present the compiler transformations that are needed to statically guarantee the *Worst Case Memory Consumption* (WCMC) and *allocation*.
— *Real-time analyzable back-end code generation and garbage collection*: We present a strategy to replace the object allocation bytecodes with real-time analyzable alternatives. Furthermore, we also present an object placement strategy, which allows for O(1) heap accesses in all cases.

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

Fig. 1: Syntax of the synchronous subset of the SystemJ language

The rest of the paper is arranged as follows: Section 2 gives the preliminary 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. Finally, we conclude in Section 8.

2. PRELIMINARIES

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 [Malik et al. 2010] extends the Java programming language by introducing both synchronous and asynchronous concurrency. SystemJ integrates the synchronous essence of the Esterel [Berry and Gonthier 1992] language with the asynchronous concurrency of *Communicating Sequential Processes* (CSP) [Hoare 1978]. SystemJ is grounded on delicate and rigorous mathematical semantics and thus is amenable to formal verification. On the top level, a SystemJ program comprises a set of clock-domains (CDs) executing asynchronously. Each clock-domain is a composition of one or more synchronous parallel reactions, 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 signals to communicate with each other. Reactions can themselves be composed of more synchronous parallel reactions, 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 Figure 1. In this work we are only interested in the synchronous subset of SystemJ.

Signals 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. 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 for a single tick, but

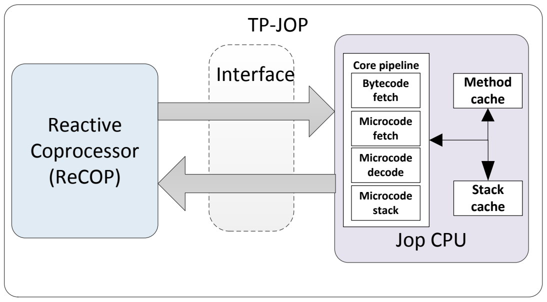


Fig. 2: Overview of the TP-JOP architecture

the value of a signal is persistent over ticks. Emission of a signal, broadcasts it across all reactions within the clock-domain. Moreover, the updated status and value of the signal, upon emission, is only visible in the next tick.

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 [1]. Furthermore, shared variable communication between synchronous parallel reactions is not allowed in SystemJ. The shared memory communication model is replaced by the signal emission (broadcast) based communication mechanism. State boundaries are demarcated explicitly by programmers using the **pause** construct. Finally, there are two types of loops in SystemJ: temporal; consisting of a **pause** statement, loops that execute forever and bounded data-loops like in standard Java.

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 multi-core time predictable execution platform called *Tandem-Java Optimized Processor* (TP-JOP) [Salcic and Malik 2013; Nadeem et al. 2011]. 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 [Park et al. 2014a] and (2) there already exists tools for statically estimating tight WCRT bounds for synchronous SystemJ programs on the TP-JOP architecture [Li et al. 2014].

Figure 2 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 [Schoberl 2003] core is a hardware implementation of the Java Virtual Machine (JVM). The JOP core is a four stage pipelined processor. Every Java bytecode is

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

translated into one or more microcodes, which are the native instruction of the JOP processor. The 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 that acts as a replacement for the data cache found in general purpose processors. The second is the method cache 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 that can lead to a method cache miss, which can be accounted for in the static WCRT analysis procedure with ease. Finally, the interface connecting the ReCOP and JOP has single clock cycle latency as described in [Salcic and Malik 2013].

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, 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 is returned back to ReCOP.

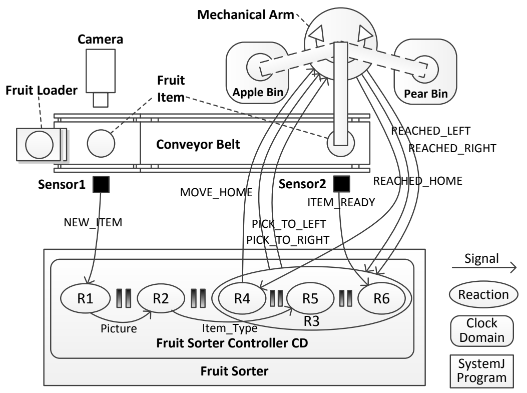
3. A MOTIVATING EXAMPLE – STATICALLY ESTIMATING WCRT OF SYSTEMJ SYNCHRONOUS CLOCK-DOMAINS

We present a motivating example of fruit sorter control system for elucidating the synchronous semantics of the SystemJ language and explaining the approach employed for static WCRT analysis of SystemJ synchronous clock-domains.

As shown in Figure 3a, the fruit sorter consists of a fruit loader, a conveyor belt, a mechanical arm, a camera and two sensors. The fruit loader places fruit items on the left end of the conveyor belt at a constant speed and the conveyor belt moves from left to right carrying fruit items. The pictorial representation of SystemJ clock-domain controlling the fruit sorter is also shown in Figure 3a. The clock-domain contains six reactions (from R1 to R6). The presence of a fruit item loaded at the left end of the conveyor belt is detected by Sensor 1 that produces an input signal `NEW_ITEM` to R1. Upon arrival of this signal, a picture of the fruit is taken in R1 and analyzed to determine the type of the item (either apple or pear in this example) in R2. This analysis is then used to sort the fruit into the appropriate bin via a Mechanical sorting arm.

Reactions R4-R6 wait for three input signals in parallel before performing actual sorting operation: (1) the `Item_Type` signal from R2, (2) `ITEM_READY` signal produced by Sensor 2, indicating that a fruit item is present at the right end of the conveyor belt and ready to be sorted and (3) `REACHED_HOME` signal indicating the mechanical arm has completed the previous sorting procedure. Upon the arrival of all these three input signals, R3 emits `PICK_TO_LEFT` or `PICK_TO_RIGHT` signal to indicate to the mechanical arm to pick the item and move to the correct bin, R3 then awaits for the `REACHED_LEFT` or `REACHED_RIGHT` signal indicating that the mechanical arm has reached atop the correct bin and dropped the item, then `MOVE_HOME` signal is emitted to bring the mechanical arm back to its initial position.

The SystemJ clock-domain implementing this control logic is shown in Figure 3b. There are two hard real-time requirements for the control logic: (1) the time spent on item recognition should be shorter than the inter-arrival time of the incoming input signal `NEW_ITEM`, else, the type of the fruit cannot be recognized and (2) the mechanical arm should sort items faster than the inter-arrival time of the input signal `ITEM_READY`, else, the fruit will not be placed into the bin and will drop off the conveyor belt. These



(a) Pictorial representation of the fruit sorter control system

```
import fruitsorter.*;
//Declare the SystemJ clock-domain
//with its interface signals
FruitSorterController(
 input signal NEW_ITEM,ITEM_READY,REACHED_HOME;
 input signal REACHED_LEFT,REACHED_RIGHT;
 output signal PICK_TO_LEFT,PICK_TO_RIGHT,
  MOVE_HOME;
)->{
 Image signal Picture;
 String signal Item_Type;
 while(true) {
  {//R1: take a picture of item using Camera
   await (NEW_ITEM);//from Sensor1
   emit Picture(Camera.takepicture());
  }
  ||
  {//R2: recognize and emit image type
   await (Picture);
   emit Item_Type(
    Process_Image.get_item_type (
    (Image)#Picture));
  }
  ||
  {//R3: Mechanical arm controller logic
   await(REACHED_HOME); //R4
   ||
   await(Item_Type); //R5
   ||
   await(ITEM_READY); //R6
   if ((String)#Item_Type.equals(``apple''))
   {//put item into apple bin
    emit PICK_TO_LEFT;
    await(REACHED_LEFT);
   } else {
    //put item into pear bin
    emit PICK_TO_RIGHT;
    await(REACHED_RIGHT);
   }
   emit MOVE_HOME;
  }
  pause;
 }
}
```

(b) Fruit sorter controller clock-domain implementation in SystemJ

Fig. 3: Fruit sorter control system

timing constraints essentially mean that the WCRT for synchronous fruit sort controller should be shorter than the inter-arrival time of input events.

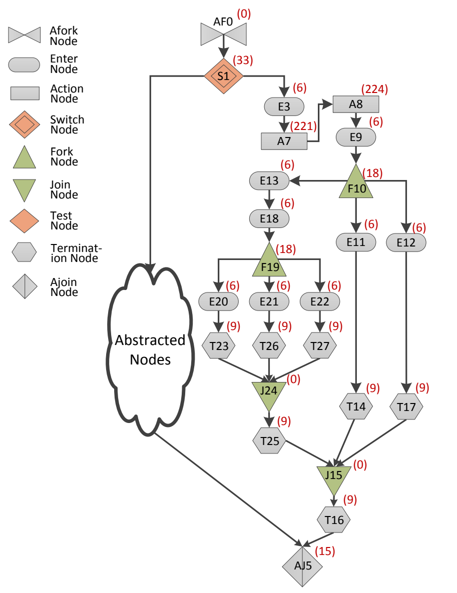
3.1. Static WCRT analysis of a SystemJ synchronous clock-domain

In order to guarantee the timing requirements of the fruit sorter control logic we need to statically determine the WCRT of the fruit sorter controller clock-domain. The static analysis performed to estimate the WCRT of any synchronous SystemJ clock-domain is described in [Li et al. 2014]. Here in we give a brief overview of the procedure.

Every SystemJ clock-domain is compiled into an intermediate format called the *Asynchronous Graph Code* (AGRC) [Malik et al. 2010]. The AGRC format of the fruit sorter controller clock-domain is shown in Figure 4a. The AGRC intermediate format is akin to the control-flow graph of a program developed in a standard programming language, except that it also encodes parallelism and state representation specific to SystemJ.

Definition 3.1. *Clock-domain transition (or tick)*: A single clock-domain transition also called a clock-domain tick is a single traversal of the AGRC graph from the root node (`AforkNode`) to the leaf node (`AjoinNode`).

Every AGRC starts with the `AforkNode` representing forking of multiple clock-domains, in case of the fruit sorter controller, a single clock-domain is forked. The `Enter` and `Switch` nodes together encode the state information. The `Enter` node sets the value of a `Switch` node in any given program transition. The `Switch` node then uses this value in the next transition to choose a branch for execution. The synchronous parallelism is captured using `Fork` and `Join` nodes. Every `Fork` node forks multiple synchronous parallel reactions for execution, which are then synchronized to the clock-domain tick at



(a) The AGRC intermediate format for the fruit sorter controller clock-domain

```
1  public class FruitSorterController {
2   //signal NEW_ITEM declaration
3   private static Signal NEW_ITEM;
4   ...//other signal and object declarations
5   public static void init () {
6    NEW_ITEM = 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 representing action node
17     //in AGRC
18     case 24: Native.wr(RESULT_REG,
19     MethodCall24_0());
20    }
21   }
22  }
23  private static boolean MethodCall24_0(){
24    // Compiled from Figure 3b, line 15
25    imm_thread_4 = Camera.takepicture();
26    //signal value emission
27    picture_1.setValue(imm_thread_4);
28    return false;
29  }
30 }
```

(b) The generated back-end Java code for JOP

Fig. 4: AGRC intermediate format and generated back-end code for TP-JOP architecture

the `Join` node. The signal emissions and Java data-computations are captured within the `Action` nodes. The `Test` nodes are used for branching on signal statuses. Finally, `Terminate` nodes, capture the termination context of the currently executing clock-domain transition. A program is considered alive and ready to execute the next tick if the AGRC pass terminates with value of 1, a value of 0 means completion of the program and all further program transitions are disabled.

The AGRC graph needs to be annotated with timing information in order to compute the WCRT of the clock-domain. All *Control Nodes* (CNs), i.e., all nodes in the AGRC, except for `Action` nodes encapsulating Java data-computations 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 *Java Data Nodes* (JDNs) are dispatched, encapsulated within individual Java methods (see Figure 4b, line 23), for execution on JOP. One needs to compute the *Worst Case Execution Time* (WCET) for each of these JDNs, using the JOP provided worst case analysis tool (WCA) [Schoeberl 2005]. The computed worst case execution times for both: CNs and JDNs are back annotated onto the AGRC, shown by the numeric annotations on the AGRC nodes in Figure 4a. Once, annotated, the AGRC model is further translated into a labeled transition system (LTS) for input into the Uppaal model-checker [Behrmann et al. 2004]. A computational tree logic (CTL) [Clarke et al. 2000] property of the form: $A[](wcrt < num)$, which, put informally, asks the model-checker to *verify* that there exists no path in the program, from the `AforkNode` to the `AjoinNode`, with a wcrt value greater than some number num. The integer value $wcrt$ is incremented by the model-checker by the WCET value of the AGRC node encountered.

This model-checking approach to computing the WCRT of a SystemJ clock-domain is essential, because SystemJ clock-domains are *open* systems, unlike standard transformational programs and hence, all possible paths need to be explored to find a tight WCRT of the SystemJ clock-domain.

4. A NEW MEMORY MANAGEMENT APPROACH FOR SYSTEMJ

The WCRT approach described above is suitable *only* for programs that do not invoke *Garbage Collection* (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 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 needs to be traversed as one of the collection phases. Puffitsch [2013] shows that a WCRT value can be obtained for programs, by pessimistically bounding the collection phase during static analysis. But, the resultant WCRT value is in seconds, orders of magnitude larger than real execution time (usually in micro-seconds), which makes garbage collection unusable for real-world applications.

One needs to include the GC time within the WCRT analysis framework of SystemJ clock-domain for the WCRT analysis to be of any practical use. The very first approach might be to include an incremental RTGC as described in Section 1, but all current RTGC proposals require preemptive scheduling, which does not bode well with the atomic tick based execution of SystemJ clock-domains. Giving up on atomicity of SystemJ clock-domain execution is not a viable option, since atomic tick based clock-domain execution is the corner stone for formal verification and WCRT analysis of synchronous programs. Of course, an RTGC might be scheduled in the slack time – time between completion of a clock-domain transition and arrival of next set of input events, but one still suffers from the problem of unpredictable GC cycle time, since a GC cycle needs to complete within the available slack.

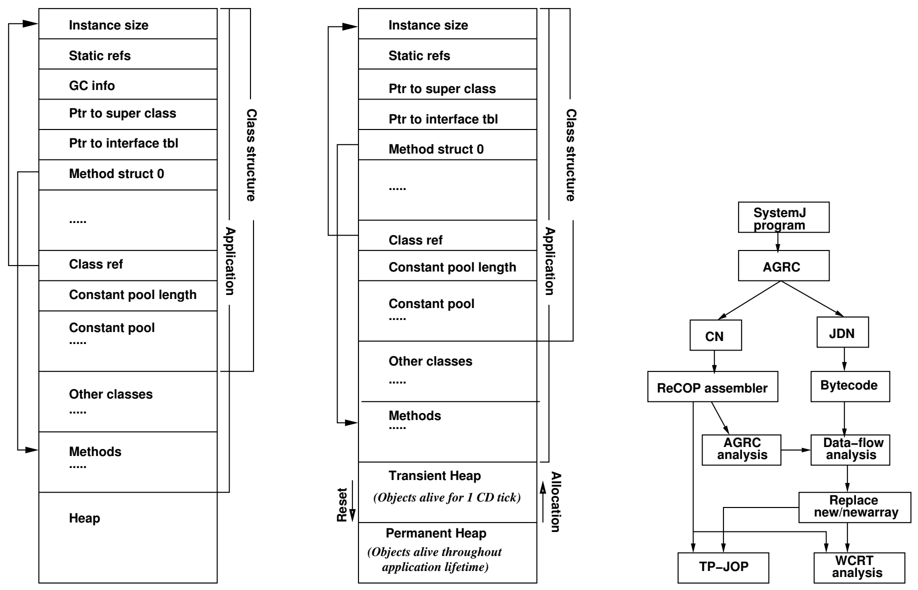
Our solution to the above problem is to simplify garbage collection. More concretely, we divide the heap space into two parts: a bounded permanent heap and a transient heap with bump pointer based object allocation and a simple pointer reset based garbage collection, which are both, time and space bounded and efficient.

Signals (valued or pure) are the only communication primitive within a SystemJ clock-domain. Upon emission of a valued signal, its status and value become visible to the other reactions in the *next* clock-domain transition. Signals, internally themselves implemented as Java objects (see Figure 4b, line 3), 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 potentially 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 transition and can be garbage collected at the end of the transition.

The basic idea is to allocate the permanent objects, we consider signals to be permanent objects as well, within a special memory area called *permanent heap* and allocate all transient objects within a *transient heap*. The transient heap gets garbage collected 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 RTGC based memory organization in Figure 5.

Both runtime memory organizations; original (Figure 5a) and proposed (Figure 5b) consists of a number of common elements required to execute every Java application such as, the constant pool, method structure table for instance method invocation,



(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 Figure 2.

references to static objects, etc. The major point of difference is that the *GC info* word 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 zeroed or overwritten at the end of the clock-domain transition. Or 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 garbage collected 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 described memory management technique.

5. STATIC ANALYSIS FOR COMPILE TIME MEMORY ALLOCATION

The complete compiler tool-chain flow in shown in Figure 5c. The SystemJ program is first compiled into the intermediate AGRC 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) AGRC analysis is used to determine the *must* and *may* Java reachable methods from any given Java program point and (2) *forward*

```
 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

```
 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

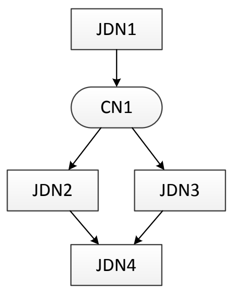
Fig. 6: Example showing the need for AGRC and data-flow analysis of a SystemJ program

and *backward* data-flow analysis is performed to find the objects and arrays that need to be allocated to the permanent heap. Finally, 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 Figure 6a. The 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 (Figure 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 call `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 the `setValue` virtual method call on the signal object. We perform data-flow analysis (ReachDef analysis [Muchnick 1997] to be exact) to find out such program points in the Java program generated for execution on JOP (e.g., Figure 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 Figure 6a, we know for sure that pro-



(a) An abstracted AGRC 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 AGRC analysis

Fig. 7: An example of AGRC analysis

gram point at line 3 *must* be executed before program point at line 11. But, in the generated Java program (Figure 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 AGRC 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 AGRC and the data-flow analysis steps in the next sections.

5.1. AGRC analysis

AGRC is a *directed graph* $G = (V, E)$, where V is the set of vertices (nodes of AGRC) 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 lambda 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 call 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 the nodes, is obtained from the AGRC. Consider a snippet of AGRC graph shown in Figure 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 AGRC analysis tool, returns the result as a S-expression as shown in Figure 7b. The resultant recursive 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 AGRC graph. For example, since `JDN1` has no parents, it's `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 (Figure 7b). A complete algorithm of our AGRC analysis is given in Appendix B.

5.2. Data-flow analysis

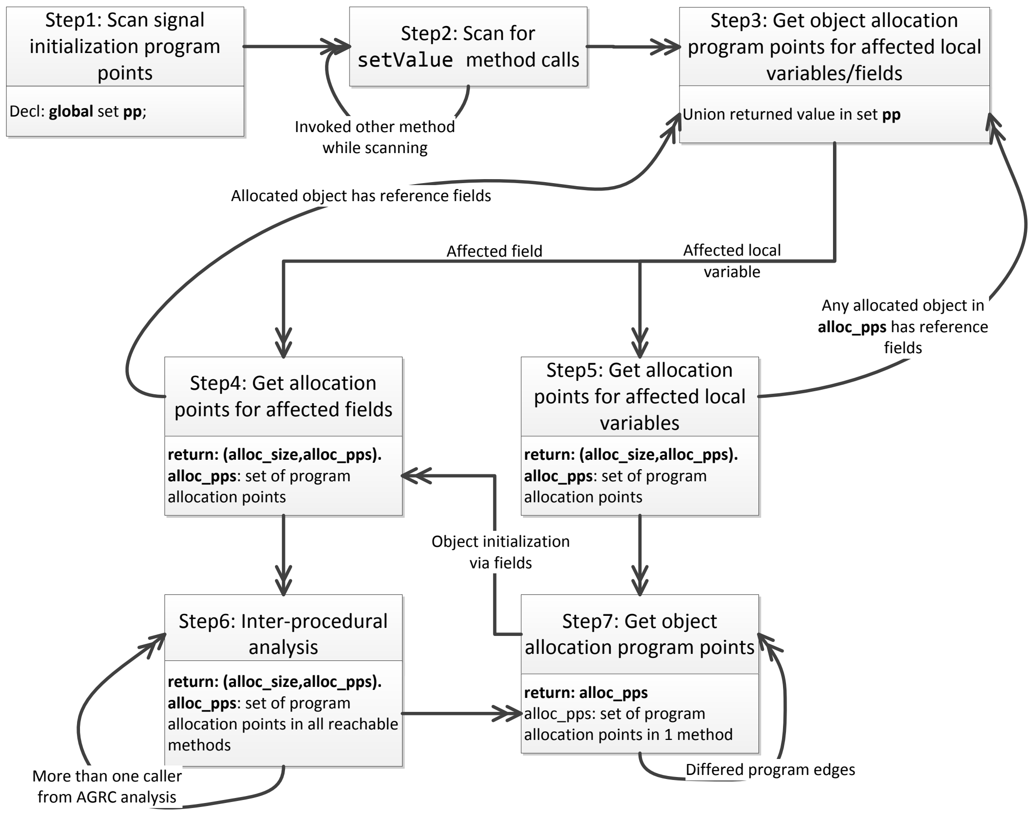

Step1: Scan signal initialization program points
Decl: global set pp;
Step2: Scan for setValue method calls
Invoked other method while scanning
Step3: Get object allocation program points for affected local variables/fields
Union returned value in set pp
Allocated object has reference fields
Affected field
Affected local variable
Any allocated object in alloc_pps has reference fields
Step4: Get allocation points for affected fields
return: (alloc_size,alloc_pps). alloc_pps: set of program allocation points
Step5: Get allocation points for affected local variables
return: (alloc_size,alloc_pps). alloc_pps: set of program allocation points
Object initialization via fields
Step6: Inter-procedural analysis
return: (alloc_size,alloc_pps). alloc_pps: set of program allocation points in all reachable methods
Step7: Get object allocation program points
return: alloc_pps
alloc_pps: set of program allocation points in 1 method
More than one caller from AGRC analysis
Differed program edges

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

Given the class file(s), the data-flow analysis carried out is shown in Figure 8. We use the flow-chart in Figure 8 to describe the steps of our data-flow analysis procedure. The detailed algorithm for each step is provided in Appendix A. Being a complex procedure we use simple code examples to describe the data-flow analysis procedure.

Consider the Java program in Figure 9a. This program consists of a single signal S (line 1). The main method (Figure 9c) initializes signal S via method init, next method MethodCall0_4 is called, which initializes objects B and C, whose types b and c are inherited from a single parent class a as shown in Figure 9d. Object t (Figure 9a, line 9) of type a is initialized to either type b or c (line 14, line 20) depending upon the Boolean a_thread_2. Finally, object t is emitted via signal S and this emitted signal value is printed within a conditional branching construct (lines 25-32).

★ Step1 of the data-flow analysis algorithm starts by scanning the Java program for program points allocating signals (e.g., method init in Figure 9a). A globally acces-

```
1   private static Signal S;
2   private static boolean a_thread_2;
3   public static void init () {
4    S = new Signal();
5   }
6   public boolean MethodCall0_4() {
7    b B = new b(33);
8    c C = new c(1001,101);
9    a t;
10   /* Use polymorphism for equality */
11   a_thread_2 = true;
12   if (a_thread_2){
13    int b1 = B.getb1();
14    t = new b(b1);/*Making deep copies*/
15    ((b)t).bbb = new Integer(99);
16   }
17   else {
18    int cc = C.getcc();
19    int ccc = C.getccc();
20    t = new c(cc,ccc);/*Making deep copies*/
21    ((c)t).mm = new Integer(999);
22   }
23   /* Emit t via S */
24   S.setValue(t);
25   if (a_thread_2)
26    /* Print */
27    System.out.println(((b)S.getValue()).bb);
28   else {
29    /* Print */
30    System.out.println(((c)S.getValue()).cc);
31    System.out.println(((c)S.getValue()).ccc);
32   }
33   return true;
34  }
```

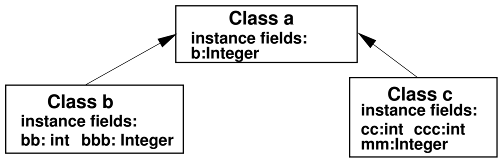
(a) Example 1: variables and intra-procedural analysis

```
1   private static Signal S;
2   private static a A;
3   public static void main(String args){
4    A = new a();
5    A.b = new Integer (70);
6    MethodCall0_0();
7   }
8   public boolean MethodCall0_0(){
9    /* Test 1 */
10   S.setValue(A);
11   return false;
12  }
```

(b) Example 2: Fields and inter-procedural analysis

```
1   public static void main (String args) {
2    init ();
3    MethodCall0_4();
4   }
```

(c) Example 1 continued: variables and intra-procedural analysis



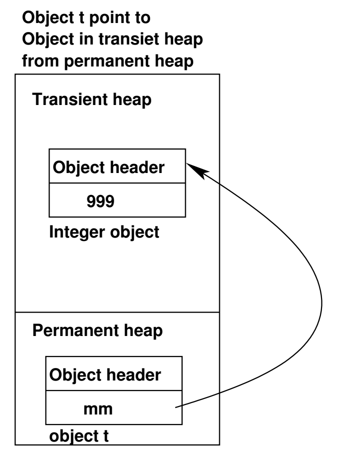
(d) UML class hierarchy for classes `a`, `b` and `c` used in Figure 9a and 9b

Fig. 9: Running example used to explain our data-flow analysis procedure

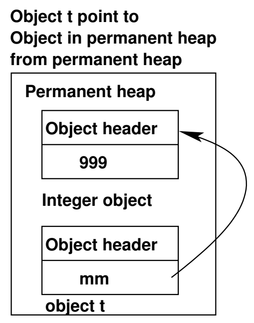
sible set pp is defined that holds all the program points allocating the objects that will be placed in the permanent heap. Step1 scans all reachable methods from the starting program point (usually the `main` method) for signal allocation and places these program points into the set pp. Each individual tuple within the set pp consists of a one or more program points that allocate objects. After the scan phase, the set pp holds $\{(5, \{\texttt{init} : 4\})\}$ for the example in Figure 9a. The first element of the tuple gives the *instance* size of the `Signal` class, the second element is the allocation program point [2] for the example in Figure 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 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 Figure 9a, this

[2]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.



(a) Pointers from permanent heap to transient heap



(b) Points-to only in permanent heap

Fig. 10: The points-to problem

involves scanning method `MethodCall0_4` and placing the program point: line 24, in set PPC. Finally, Step3 of the algorithm is called to obtain all object allocation program points whose returned references are emitted via signals.

★ Step3 analyzes the emission expression for every program point in set PPC. A emitting expression can *only* ever emit a method local variable (as in the case of Figure 9a) or a field (as in the case of our soon to come example in Figure 9b). In our current example a variable is being emitted (or formally called 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 line 24, $alloc\_pps = \{\texttt{MethodCall0\_4} : 14, \texttt{MethodCall0\_4} : 20\}$. Looking at Figure 9a, it is clear that object reference `t` can be initialized at either line 14 or line 20, depending upon the value of variable `a_thread_2` during program execution. For each of these program points, Step5 guarantees that the object reference does not *escape* the lifetime of the method [Choi et al. 1999]. [3] 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, Step3 is invoked for migrating all object allocations whose returned reference is held in these fields into permanent heap. Consider, the program point on line 20, object `t` is initialized as class `c` (for program point 14, `t` is initialized and analyzed as class `b`), which has three fields: two (`cc` and `ccc`) are primitive fields, but the third: `mm` is a reference field (of type `Integer`). The `mm` reference field is initialized to an `Integer` object at program point 21. Figure 10a shows the address held in `mm` if Step3 is not called from Step5. Reference field `mm` 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

[3]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.

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 Figure 10b.
The final computation that Step5 performs is computing the size, which will be reserved in the permanent heap, of object `t`, given that $alloc\_pps$ may have multiple (two in our current example) elements. The primary idea is to reserve the maximum size amongst all different types being initialized. For the running example, `t` can be of type `b` or `c` during program execution due to polymorphism. We compute the *instance* size of both these types as 3 and 4 words, respectively [4] and reserve 4 words (plus object header size) in the permanent heap space. Step7 returns multiple program points in set $alloc\_pps$ iff 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 consciously decided to not allow method local object references from escaping because, it leads to trade-off between space utilization and functional correctness. If we allow, for example `t` in Figure 9a, to escape method `MethodCall0_4`, then any of the reference fields of `t` (e.g., `mm`) might be reinitialized at some other program point. This 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 [Muchnick 1997] to obtain the program points defining the variable. If the defining program points are `new` or `newarray` bytecodes [Meyer and Downing 1997], 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 [Burke et al. 1995] 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. If the defining program point affects a field, then Step4 is called, which calls Step7 via Step6, we describe Step4 and Step6 next. 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 to return 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 dataflow algorithm 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` and `newarray` bytecodes with a different set of bytecodes (c.f. Section 6). We cannot blindly replace these bytecodes in methods other than the ones being

[4]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.

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 Figure 11.
In Figure 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 Figure 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, Figure 11c, line 1. If we were to replace this `new` bytecode, to allocate some memory in the permanent heap, then both; `A` and `B` will point to the same place in permanent heap. Consequently, first a constant 0 will be written in the allocated memory (Figure 11b line 3) and then this value will be overwritten to 1 (Figure 11b, line 5), result is 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, all our benchmarks have 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 Figure 9a line 14.

★ 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 Step5 calls Step7 via Step6, which performs 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 affected field being analyzed is not allocated in the same method. Let us consider the simple example in Figure 9b to explain the inter-procedural analysis. The `main` method initializes two objects: object `A` of type `a` (c.f. Figure 9d) and its `Integer` type field `b` and finally calls method `MethodCall0_0`. Field `A` is emitted via signal `S`. Our data-flow analysis algorithm needs to locate the program points at lines 4 and 5, so that these object allocations can be moved to the permanent heap space.
Part-1 of Step6 calls Step7 passing it method `MethodCall0_0` to find out the object `A` and its field's allocation program points. Step7 being an *intra-procedural* anal-

```
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

ysis 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 AGRC-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 Figure 9b, there is a single caller: the `main` method in the call-graph for callee `MethodCall0_0`, 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 AGRC-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 `new` and `newarray` 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 Figures 12a and 12b, respectively. The memory allocation algorithm (Figure 12a) takes as input the maximum memory address, the program points to replace and the whole program itself as input. Two variables: $allocPtr$ and $header\_size$ are initialized to the 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, Figure 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 7). 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 Figure 12b.

The algorithm (Figure 12b) used to replace the object allocation bytecodes takes as input three arguments: the allocation pointer, $allocPtr$, the program point to replace pp, and the method containing the program point: M. There are *two* types of object allocation bytecodes as specified in the *Java Virtual Machine* (JVM) specification [Meyer and Downing 1997]: the `new` and `newarray` bytecodes. Moreover, the `newarray` bytecode itself is of two varieties, dependent upon the type of array being allocated: (1) the `newarray` bytecode that allocates space for arrays holding primitive values and (2) the `newarray` bytecode that allocates space for arrays holding object references. This distinction is important since the number of bytes used to represent each of these bytecode varies. The `new` and `newrray` bytecodes that operate on object reference types are encoded in the class file with three bytes, whereas the `newarray` bytecode that allocates arrays of primitive type only uses two bytes in the class file representation.

Different methods are called to replace each of these allocation variants: Figure 12b, line 3 replacing `new`, line 8 replacing `newarray` allocating arrays of primitive types, and line 9 replacing `newarray` allocating arrays of object references. All methods pretty much do the same thing, except that some padding (represented as `nop` bytecodes) is needed in case of the two variants of the `newarray` bytecodes. This padding is

```
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
let $allocPtr \leftarrow$ maxMem;
let $header\_size \leftarrow 2$;
foreach $(size, dd) \in defs$ do
    let $dd \leftarrow$ sortUniqueDescending $(dd)$;
    let $M \leftarrow$ load $(dd)$;
    let $allocPtr \leftarrow allocPtr - (size + header\_size)$;
    if $allocPtr < 0$ then raise No_mem;
    foreach $d \in dd$ do
        replaceByteCode $(allocPtr, d, M)$;
    end
end
```

(a) Bump-pointer memory allocation

```
Input: $allocPtr$: allocation pointer
Input: pp: program point to replace
Input: $M$: Method containing the program point
1  if $pp$ = new then
2      let $arg \leftarrow$ getArg $(pp)$;
3      replaceNew $(pp, M, arg)$;
4      adjustConditionals $(M, 22)$;
5  else if $pp$ = newarray then
6      let $arg \leftarrow$ getArg $(pp)$;
7      if $arg$ = primitive then
8          replaceNewArrayB $(pp, M, arg)$;
9      else replaceNewArrayO $(pp, M, arg)$;
10     adjustConditionals $(M, 22)$;
11 else raise error
```

(b) Replacing the new/newarray bytecodes

Fig. 12: The back-end code generation pseudo-algorithm

needed to correctly update the conditional instruction's target address after replacement (line 10).

All conditional instruction's, following the object allocation bytecodes, target addresses need to be incremented by 22 bytes after replacement. This 22 byte increment is made obvious by Figures 12c and 12d, which show the Java bytecode snippets for the example program in Figure 9b before and after new bytecode replacement, respectively [5]. Figure 12c shows the bytecode produced by the Java compiler for the A object allocation in the main method. The very first bytecode, new, takes as argument the index, 2, into the constant pool that holds the type of object to allocate. The returned object reference is then duplicated (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, 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 timing predictable bytecodes as shown in Figure 12d.

A single new bytecode (consuming 3 bytes in the class file) is replaced with a list of bytecodes lines 9-21 (line numbers are given on the right) consuming 25 bytes in the class file, hence the 22 byte increment of target addresses of the conditional bytecodes. The core idea is very simple; since we have already allocated memory for the object (Figure 12a), 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 Figure 12e), 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 Figure 12d. The second word of the object header is the pointer to the start of the method structure of the class (c.f. Figure 5b). The second word of the object header is initialized in lines 14-21 in Figure 12d.

In case of newarray bytecode, the array object structure (shown in Figure 12f) differs from standard real-time JVM implementations. For array allocation, we again use a

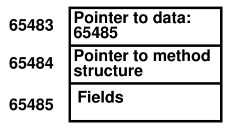
[5]In both these figures, the line numbers are shown on the right of the program. The numbers adjacent to the bytecodes are the starting bytes for each of the bytecodes as obtained by the javap -c -v command

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

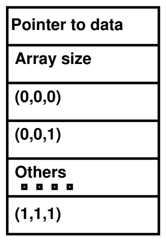
(c) Java bytecode snippet and class constant-pool for example in Figure 9b

```
--- constant-pool snippet ----
#2 = Class     #79     //a/a
#3 = Methodref #2.#78  //a/a.''<init>'':()V
#4 = Fieldref  #12.#80 //test/test.A:La/a;
.... //other constants
#154 = Integer           65483
#155 = Integer           65485
---------- main method snippet --------
 0:ldc_w         #154 // int 65483
 3:dup
 4:ldc_w         #155 // int 65485
 7:swap
 8:invokestatic  #151 // Method Native.wr:(II)V
11:dup
12:bipush        1
14:iadd
15:ldc_w         #2   // class a/a
18:bipush        5
20:iadd
21:swap
22:invokestatic  #151 // Method Native.wr:(II)V
25:dup
26:invokespecial #3   // Method a/a.''<init>'':()V
29:putstatic     #4   // Field A:La/a;
```

(d) Java bytecode snippet and class constant-pool for example in Figure 9b after `new` bytecode replacement



(e) Java object allocation in permanent heap



(f) Java (2x2x2) array allocation in permanent heap

Fig. 12: The back-end generated code and compacted object representation in heap

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 (usually 8 words), which uses spines [Pizlo et al. 2010] or handles [Schoeberl 2005] 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 irrespective of the array dimensions (as in ‘C’) rather than using linked lists for array traversal in order to allow $O(1)$ heap accesses.

7. EXPERIMENTAL RESULTS

We have carried out two sets of experiments: micro-benchmarks and static WCRT analysis on well established real-time benchmarks to study the efficacy of the proposed memory management scheme compared to two different real-time garbage collector (RTGC) implementations [Schoeberl 2005; Pathirana 2013]. The first is the RTGC proposed in [Schoeberl 2005], which is a concurrent version of Cheney’s copy collector developed by Baker [Baker Jr 1978]. In the rest of the section we call it JOP-GC. The second one proposed in [Pathirana 2013] is a variation of the implementation of the allocation and real-time collector described in [Pizlo et al. 2010], in the rest of the section we term it AGC. All our micro-benchmark experiments are run in ModelSim, simulat-

Table I: 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)
100	28	2145	2995
200	28	4290	5590
400	28	8580	11180
500	28	10725	13975
1000	28	21450	27950
2000	28	42900	55900
4000	28	85800	111800

ing 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.

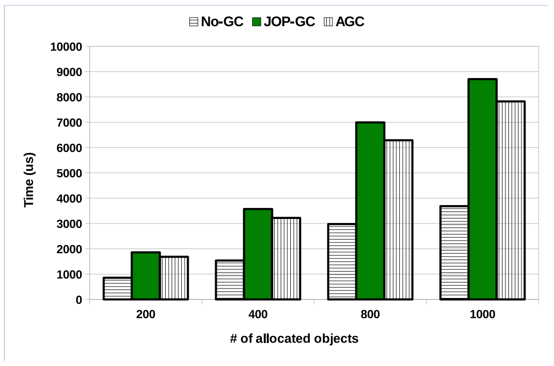
7.1. Micro benchmarks

We carry out 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 are allocated and the second object's reference is set as the array value. The example benchmarks are available from [Pathirana 2013].

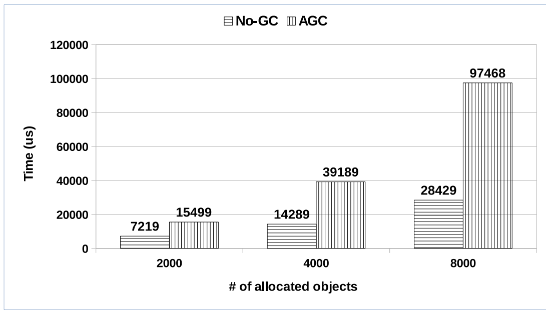
The runtime comparison between the three approaches for each of the micro-benchmarks is shown in Figure 13. Figure 13 gives time for completion of a memory allocation request in two cases: (1) when the GC is not invoked, i.e., there is enough memory available to allocate the requested space and (2) when the GC is invoked to collect the garbage when enough memory is not available upon a memory allocation request from the mutator. The proposed approach is, on average, approximately $3\times$ faster compared to the RTGC based approaches [6]. 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. In case of memory allocation time when no GC is invoked, there are multiple reasons for the speedup: (1) there are no read/write barriers when allocating memory, which are needed by both: JOP-GC and AGC, since RTGCs are preemptable. (2) The current `new` and `newarray` 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. These runtime numbers are important, because they influence the WCRT as we will see in the next section. (3) In case of `newarray` bytecode execution, array spines need to be initialized for AGC [Pizlo et al. 2008; Pathirana 2013], which leads to further slow down in memory allocation times.

Table I gives the number of words allocated for the complex object allocation benchmark for all the three approaches. In the proposed approach, every iteration of the loop reuses the space allocated on the permanent heap space (same allocation figures would be achieved if the program was recursive rather than iterative). The GC approches on the other hand, allocate memory on each `new`/`newarray` bytecode execution. 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 al-

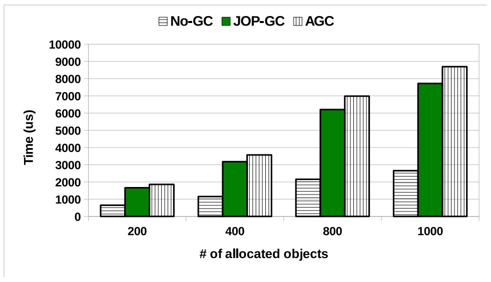
[6]In some cases JOP-GC could not perform garbage collection correctly, as it ran out of handles. Hence, we have only compared with AGC in such cases.



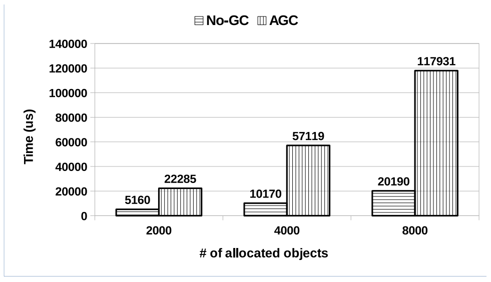
(a) Simple object memory allocation runtime without GC



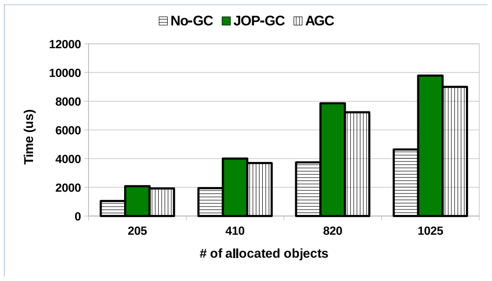
(b) Simple object memory allocation runtime with GC



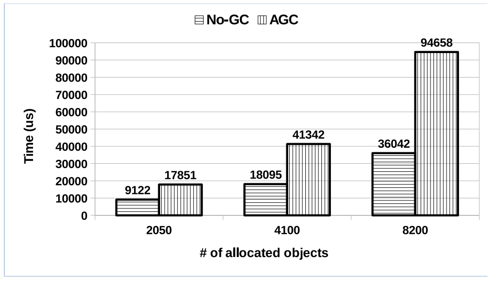
(c) Simple array memory allocation runtime without GC



(d) Simple array memory allocation runtime with GC



(e) Complex object memory allocation runtime without GC



(f) Complex object memory allocation runtime with GC

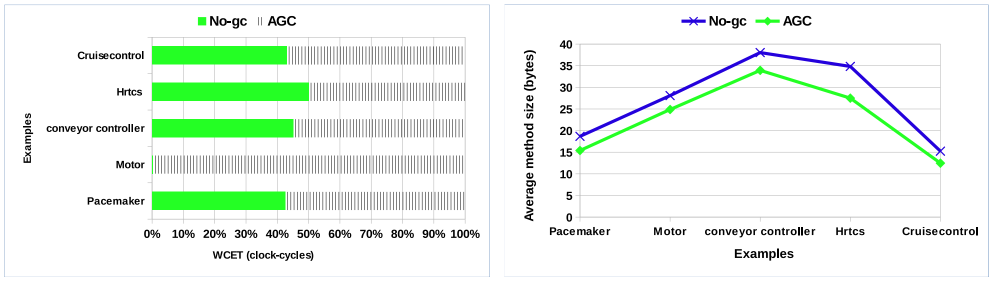
Fig. 13: Runtime for micro-benchmarks

location, which leads to internal memory fragmentation. Whereas JOP-GC uses object replication strategy with semi-space partitioning [7].

7.2. WCRT comparisons

In this section we compare the static WCRT obtained by applying the technique described in Section 3.1 on a set of real-time benchmarks. To keep ourselves honest, we chose the benchmarks whose memory allocation requests fit within the allocated heap space without GC invocation. Consequently, we also hand-deleted all code related to GC. This was a necessary step, because one cannot *practically* bound the GC cycle time

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



(a) Statically analyzed WCET values for benchmarks without GC invocation (b) Statically analyzed method sizes for benchmarks

Fig. 14: WCET and method size comparison: No-GC vs. AGC

for purpose of static analysis. A pessimistic GC cycle time is in seconds [Puffitsch 2013], which is orders of magnitude larger than the WCRT value of the proposed approach.

We chose 5 real-time synchronous programs for bench-marking. `Cruise control` [Charles 1996] is the cruise speed controller found in cars. `Hrtcs` [Park et al. 2014b] is a human response time gathering system. `Conveyor controller` is the motivating example described in this paper. `Motor` [Bourke12 and Sowmya 2009] is a stepper motor control system for a printer. Finally, `Pacemaker` [Park et al. 2014a], 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. The results are shown in Figure 14a. 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, Figure 14b gives the average method size comparisons for the compiled SystemJ programs. Since we replace a single object allocation bytecode `new` or `newarray` 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 `new`/`newarray` bytecode execution in the AGC approach.

8. CONCLUSION AND FUTURE WORK

In this paper we have presented a new memory organization and model for hard real-time and safety-critical systems programmed with managed language runtimes, primarily Java. The presented approach takes a departure from the current practice of building new memory allocation and garbage collection algorithms that can meet at best soft real-time guarantees. The presented approach utilizes the formal synchronous programming model provided by SystemJ, to divide 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. (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.

It is impossible to bound the garbage collection cycle times to values that are of practical use – primarily due to the linked list traversal needed in different phases of the garbage collection, and hence, preemptive and incremental garbage collection approaches have been the only solution to programming real-time systems with managed runtime environments to date. Preemptive scheduling generally leads to explosion in the number of states that need to be checked by automated formal verification tools, and hence, makes building safety critical systems hard, if not impossible. This proposal provides a solution to this problem. All the aforementioned qualitative improvements are further accompanied by improvements in the program throughput (approximately $3\times$ 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. Not to mention the improvements in the number, size and compacted layout of allocated Java objects and arrays.

However, the proposed approach is not a penance. The proposed memory management technique, does indeed require the programmer to carefully design their real-time applications, as deep object copies need to be made explicitly by the programmer. Furthermore, we see many opportunities for future optimization, especially getting rid of the object escapement restrictions currently enforced by the proposed data-flow analysis. We also see an opportunity to *reuse* permanent memory heap allocations across clock-domain tick boundaries, which we plan to address in the future.

REFERENCES

2013. JSR 302: Safety Critical Java Technology. http://jcp.org/en/jsr/detail?id=302. (2013).

David F Bacon, Perry Cheng, and VT Rajan. 2003. A real-time garbage collector with low overhead and consistent utilization. *ACM SIGPLAN Notices* 38, 1 (2003), 285–298.

Henry G Baker Jr. 1978. List processing in real time on a serial computer. *Commun. ACM* 21, 4 (1978), 280–294.

Gerd Behrmann, Alexandre David, and Kim G. Larsen. 2004. A Tutorial on UPPAAL. In *International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures (Lecture Notes in Computer Science)*, M. Bernardo and F. Corradini (Eds.), Vol. 3185. Springer Verlag, 200–237. http://doc.utwente.nl/51010/

G. Berry. 1993. The semantics of pure esterel. (1993). citeseer.ist.psu.edu/berry93semantics.html

G. Berry and G. Gonthier. 1992. The Esterel Synchronous Programming Language: design, semantics and implementation. *Science of Computer Programming* 19, 2 (1992), 87–152.

Amar Bouali. 1998. XEVE, an ESTEREL verification environment. In *Computer Aided Verification*. Springer, 500–504.

T Bourke12 and A Sowmya. 2009. Delays in Esterel. *SYNCHRON09* (2009), 55.

Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. 1995. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Languages and Compilers for Parallel Computing*. Springer, 234–250.

André Charles. 1996. Representation and analysis of reactive behaviors: A synchronous approach. In *Computational Engineering in Systems Applications, CESA*, Vol. 96. 19–29.

Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Sreedhar, and Sam Midkiff. 1999. Escape analysis for Java. *Acm Sigplan Notices* 34, 10 (1999), 1–19.

E. M. Clarke, O. Grumberg, and D. Peled. 2000. *Model Checking*. MIT Press.

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous dataflow programming language Lustre. *Proc. IEEE* 79, 9 (September 1991), 1305–1320.

Klaus Havelund and Thomas Pressburger. 2000. Model Checking JAVA Programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer* 2, 4 (2000), 366–381. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.250

Roger Henriksson. 1998. *Scheduling garbage collection in embedded systems*. Ph.D. Dissertation. Lund University.

C.A.R. Hoare. 1978. Communicating Sequential Processes. *Communication of the ACM* 21, 8 (1978), 666–677.

Tomas Kalibera, Filip Pizlo, Antony L Hosking, and Jan Vitek. 2011. Scheduling real-time garbage collection on uniprocessors. *ACM Transactions on Computer Systems (TOCS)* 29, 3 (2011), 8.

P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le. Marie. 1991. Programming real-time applications with SIGNAL. *Proc. IEEE* 79, 9 (September 1991), 1321–1336.

Zhenmin Li, Avinash Malik, and Zoran A. Salcic. 2014. 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*. 1–8. DOI: http://dx.doi.org/10.1109/RTCSA.2014.6910556

Avinash Malik, Zoran Salcic, Christopher Chong, and Salman Javed. 2012. System-level approach to the design of a smart distributed surveillance system using systemj. *ACM Trans. Embedded Comput. Syst.* 11, 4 (2012), 77. DOI: http://dx.doi.org/10.1145/2362336.2362344

Avinash Malik, Zoran Salcic, Partha S. Roop, and Alain Girault. 2010. SystemJ: A GALS language for system level design. *Elsevier Journal of Computer Languages, Systems and Structures* 36, 4 (December 2010), 317–344.

Jon Meyer and Troy Downing. 1997. *Java virtual machine*. O'Reilly & Associates, Inc.

Steven S. Muchnick. 1997. *Advanced compiler design implementation*. Morgan Kaufmann.

Muhammad Nadeem, Morteza Biglari-Abhari, and Zoran Salcic. 2011. RJOP: a customized Java processor for reactive embedded systems. In *Proceedings of the 48th Design Automation Conference*. ACM, 1038–1043.

HeeJong Park, Avinash Malik, Muhammad Nadeem, and Zoran A. Salcic. 2014a. 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*. 37. DOI: http://dx.doi.org/10.1145/2661020.2661030

Heejong Park, Avinash Malik, and Zoran Salcic. 2014b. 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*.

Isuru Pathirana. 2013. *AucklandGC: a real-time garbage collector for the Java optimized processor*. Master's thesis. University of Auckland.

Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. 2008. A study of concurrent real-time garbage collectors. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 33–44.

Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L Hosking, Ethan Blanton, and Jan Vitek. 2010. Schism: fragmentation-tolerant real-time garbage collection. In *ACM Sigplan Notices*, Vol. 45. ACM, 146–159.

Wolfgang Puffitsch. 2013. Design and analysis of a hard real-time garbage collector for a Java chip multiprocessor. *Concurrency and Computation: Practice and Experience* 25, 16 (2013), 2269–2289.

Zoran Salcic and Avinash Malik. 2013. GALS-HMP: A heterogeneous multiprocessor for embedded applications. *ACM Trans. Embedded Comput. Syst.* 12, 1s (2013), 58. DOI: http://dx.doi.org/10.1145/2435227.2435254

Martin Schoberl. 2003. JOP: A java optimized processor. In *Workshop on Java Technologies for Real-Time and Embedded Systems*.

Martin Schoeberl. 2005. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. Ph.D. Dissertation. Vienna University of Technology. http://www.jopdesign.com/thesis/thesis.pdf

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. on Embedded Computing Sys.* 7, 3 (2008), 1–53. DOI: http://dx.doi.org/10.1145/1347375.1347389