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

Journal:	*Concurrency and Computation: Practice and Experience*
Manuscript ID	Draft
Editor Selection:	Prof. David Walker
Wiley - Manuscript type:	Research Article
Date Submitted by the Author:	n/a
Complete List of Authors:	Malik, Avinash; University of Auckland, Electrical and Computer Engineering Park, Heejong; University of Auckland, Electrical and Computer Engineering Nadeem, Muhammad; University of Auckland, Electrical and Computer Engineering Salcic, Zoran; University of Auckland, Electrical and Computer Engineering
Keywords:	Real-time tasks, Static analysis, Java, Garbage collection, SystemJ

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **00**:1–23
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

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

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

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

SUMMARY

Arguably the major hurdle in adoption of memory managed languages, such as Java, in the real-time systems community is the incorporation of garbage collection (GC) within the real-time application. It has recently been shown that the statically estimated worst case execution time (WCET) value of tasks invoking GC is pessimistic to such a great extent that incorporating them within the hard real-time scheduling framework is unfeasible. 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 large pessimistic WCET bounds obtained due to GC cycle times. 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 real-time GC approaches.
Copyright © 2016 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Compiler; Static Analysis; WCET; SystemJ; Garbage Collection

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 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 derivation 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 *reacts instantaneously* 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 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 time, one needs to find out the *Worst*

*Correspondence to: avinash.malik@auckland.ac.nz, Department of Electrical and Computer Engineering, University of Auckland, NZ

Case Reaction Time (WCRT) of any given synchronous program. This WCRT value determines the shortest inter-arrival time for input events from the environment.

Techniques exist for computing WCRT values for 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 synchronous programs that support data computation via managed-languages like Java, which provides a well defined operational semantics. 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 estimates for real-time tasks with GC are very pessimistic [6].

The main **contribution** presented in this paper is a new memory management approach that is amenable to static WCRT analysis of synchronous programs intertwined with managed runtime environments for data-computation. More concretely our contributions are as follows:

- *Programming model inspired memory organization*: In this paper we present a new memory organization for Java and an accompanying static WCRT analysis based on the synchronous 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.

We use the SystemJ [3] programming language, because it provides the synchronous MoC as a subset along with Java data-driven computation support. Furthermore, there exist efficient and real-time analyzable execution architectures [7] and WCRT analysis tools [8] that support subset of SystemJ programs – those without GC invocation.

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. 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. 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 [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) [9]. 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 Figure 2a for explaining the SystemJ MoC. On the top level, a SystemJ program comprises a set of clock-domains (CD1, CD2, CD3) executing asynchronously. Each clock-domain is a composition of one or more synchronous

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

Figure 1. Syntax of the synchronous subset of the SystemJ language

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, 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, i.e., a single clock-domain.

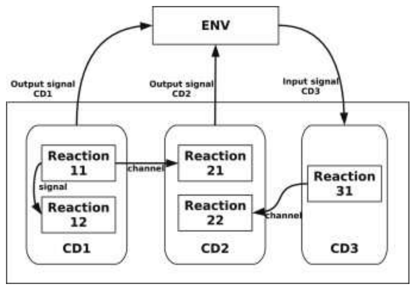
Signals are the primary means of communication between reactions of a clock-domain and between a clock-domain and its environment. A signal in SystemJ can be typed or untyped. Every untyped signal in SystemJ is called a pure signal and only consists of a Boolean status, which can be set to `true` or `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 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 [†]. 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: temporal; consisting of a **pause** statement, loops that execute forever and bounded data-loops like in standard Java.

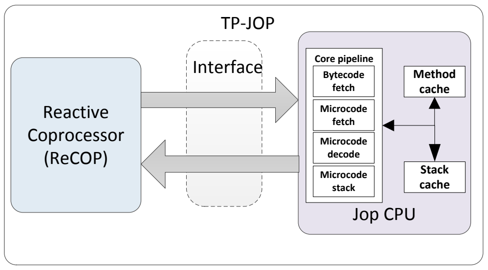
2.2. *The real-time execution architecture*

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) [10, 7]. There are two main reasons we chose this

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



(a) A graphical representation of a SystemJ program



(b) Overview of the TP-JOP architecture

Figure 2. A generic SystemJ program and its execution architecture

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 [11] and (2) there already exists tools for statically estimating tight WCRT bounds for synchronous SystemJ programs executing on the TP-JOP architecture [8].

Figure 2b 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 [12] 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 microcodes, 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 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 [10].

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. STATICALLY ESTIMATING WCRT 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 SystemJ clock-domains. 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 [8] for a more detailed description of the technique.

Figure 3a shows a simple SystemJ synchronous program. The program 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, 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.

3.1. GRC representation of the clock-domain

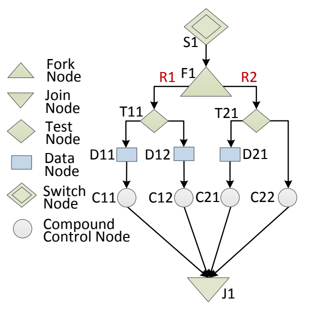
The semantics of a synchronous program described in SystemJ is captured using an intermediate format called *GRaph Code* (GRC) [3]. The GRC is a directed acyclic graph that explicitly describes the control flow of a synchronous program through primitive nodes.

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 (node with no incoming edges) to the sink node (node with no outgoing edges).

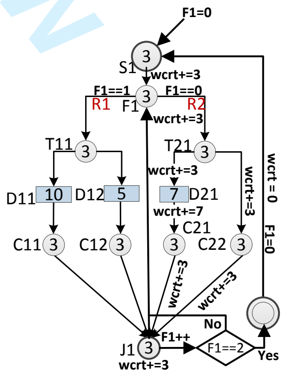
Java statements in SystemJ are represented as data nodes. State encoding and selection are captured with enter and switch nodes, respectively. Present statements are denoted by test nodes. Concurrency embodied by parallel reactions is delineated using fork and join nodes. A fork node spawns parallel 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. The GRC of the SystemJ program shown in Figure 3a is depicted in Figure 3b. 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. Finally, the infinite loops are simply infinite executions of the GRC

```
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) Example synchronous SystemJ clock-domain



(b) GRC of the SystemJ clock-domain in Figure 3a



(c) Uppaal representation of GRC

Figure 3. Example SystemJ program with its GRC and Uppaal representations

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 source node to the sink node in the GRC. Finding this longest path is non-trivial, because of 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. We use the Uppaal [13] model-checker to perform an exhaustive state space exploration of the GRC, in the process, emulating the execution of the GRC on the TP-JOP architecture to find the WCRT of the SystemJ clock-domain. The exhaustive search space carried out by the Uppaal model-checker gives WCRT estimates, for SystemJ programs, that are on average $\approx 30\%$ tighter than those obtained by other approaches [8].

There are three steps involved in finding the WCRT of a clock-domain using Uppaal:

1. *Annotating the GRC nodes with Worst Case Execution Time (WCET) values*: The 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 *Worst Case Execution Time* (WCET) for each of these JDNs, using the JOP provided worst case analysis tool (WCA) [14]. The computed worst case execution times for both: CNs and JDNs are back annotated onto the GRC. The numerical annotations on the nodes in Figure 3c show these back annotated WCETs.

2. *Translating the GRC into an Uppaal timed automaton*: Figure 3c 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., $F1$ in Figure 3c. 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 ($F1$ in Figure 3c). 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 Figure 3c). Thereby, guaranteeing sequential execution of the synchronous parallel reactions. An integer type variable $wcrt$, is added to the TA [‡]. 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 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 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 GRC to 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

[‡] 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.

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. 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 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 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 needs 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 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 with bump pointer based object allocation and a simple pointer reset based memory reclaim, which are both, 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:

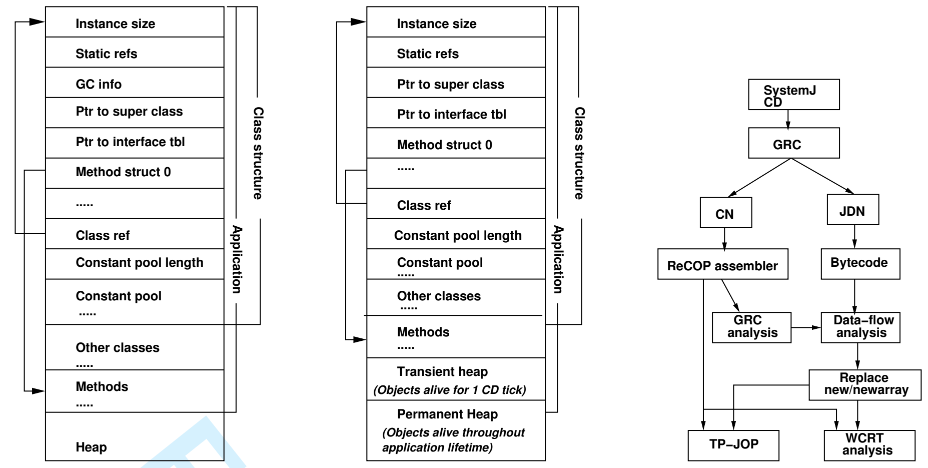
- *Permanent objects*: Java objects that are emitted via signals. These objects like signals are potentially alive throughout the application lifetime.
- *Transient objects*: Java objects that are created internally for computation, but are never emitted via signals. Transient objects are only alive during a single clock-domain tick transition and can be garbage collected 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 *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 Figure 4.

Both runtime memory organizations; original (Figure 4a) and proposed (Figure 4b) 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, 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 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.



(a) Original JOP runtime memory organization (b) Proposed JOP runtime memory organization (c) The tool-chain flow

Figure 4. 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.

5. STATIC ANALYSIS FOR COMPILE TIME MEMORY ALLOCATION

The complete compiler tool-chain flow is shown in Figure 4c. 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, 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 5a. 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 (Figure 5b). 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 [15] to be exact) to find out such program points in the Java program generated for execution on JOP (e.g., Figure 5b). 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 5a, we know for sure that program point at line 3 *must* be executed

```
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˜5a)
18     case 1: Native.wr(RESULT_REG,
19     MethodCall1_0());
20     //Call method for
21     //lines 11 - 12 (Figure 5a)
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 5a
29    a = a + 1;//line 3, Figure 5a
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

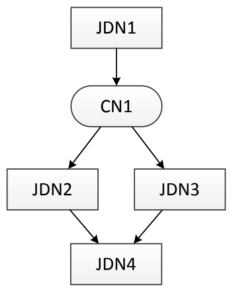
Figure 5. Example showing the need for GRC and data-flow analysis of a SystemJ program

before program point at line 11. But, in the generated Java program (Figure 5b), 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 next 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 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 GRC. Consider a snippet of GRC graph shown in Figure 6a. 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 Figure 6b. 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



(a) An abstracted GRC graph

```
(mList
 (mNode JDN1 (Must Null) (May Null))
 (mNode JDN2
  (Must (mNode JDN1 (Must Null) (May Null)))
  (May Null))
 (mNode JDN3
  (Must (mNode JDN1 (Must Null) (May Null)))
  (May Null))
 (mNode JDN4 (Must Null)
  (May
   (mNode JDN2
    (Must (mNode JDN1 (Must Null) (May Null)))
    (May Null))
   (mNode JDN3
    (Must (mNode JDN1 (Must Null) (May Null)))
    (May Null))))
```

(b) An output of the GRC analysis

Figure 6. An example of GRC analysis

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, 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 6b). A complete algorithm of our GRC analysis is given in [16].

5.2. *Data-flow analysis*

Given the class file(s), the data-flow analysis carried out is shown in the flow-chart in Figure 7. The detailed algorithm for each step is provided in [16]. 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*

Figure 8a 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`. Figure 8b 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 `B` and `C`, whose types `b` and `c` are inherited from a single parent class `a` as shown in Figure 9a. Object `t` (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 `t` 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 Figure 8a). 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 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 Figure 8a. The first element of the tuple gives

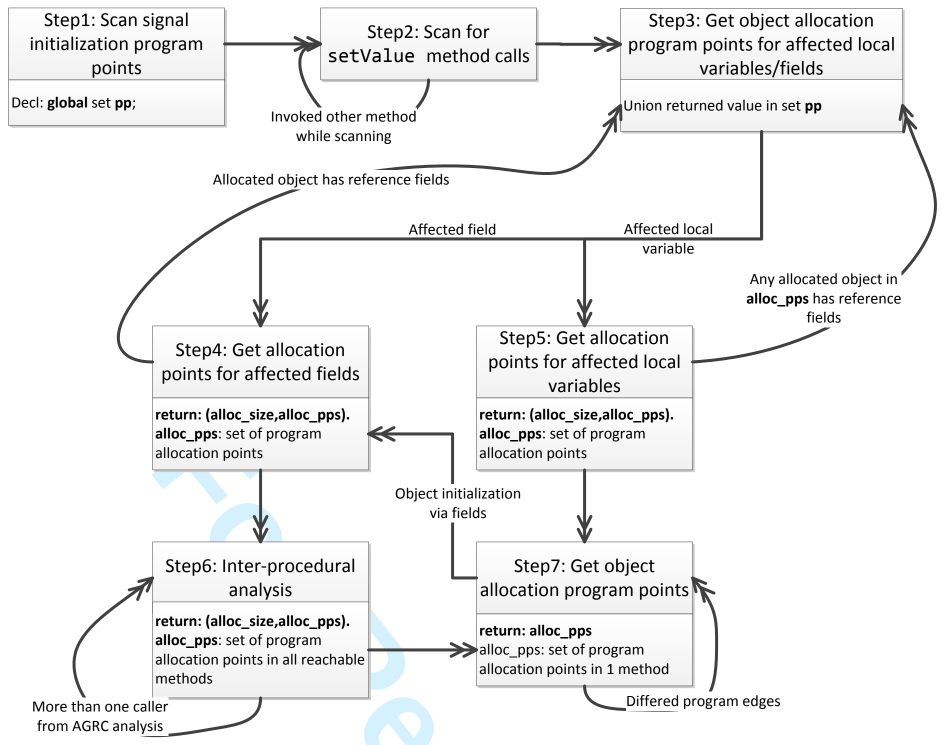
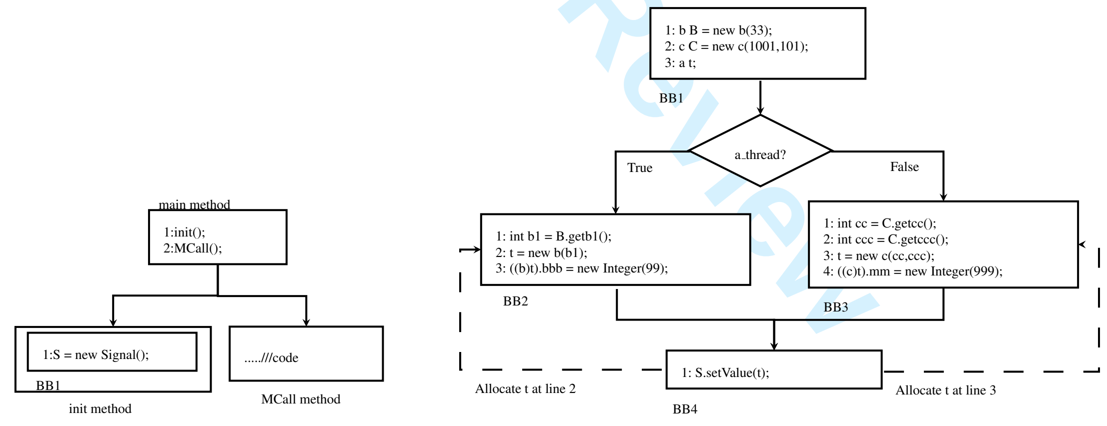

Step1: Scan signal initialization program points
Decl: **global** set **pp**;
Step2: Scan for `setValue` method calls
Invoked other method while scanning
Step3: Get object allocation program points for affected local variables/fields
Union returned value in set **pp**
Allocated object has reference fields
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
More than one caller from AGRC analysis
Step7: Get object allocation program points
return: alloc_pps alloc_pps: set of program allocation points in 1 method
Differed program edges

Figure 7. The flow-chart for the data-flow analysis algorithm



```
1:init();
2:MCall();
```

```
1:S = new Signal();
```

```
.....///code
```

```
1: b B = new b(33);
2: c C = new c(1001,101);
3: a t;
```

```
1: int b1 = B.getb1();
2: t = new b(b1);
3: ((b)t).bbb = new Integer(99);
```

```
1: int cc = C.getcc();
2: int ccc = C.getccc();
3: t = new c(cc,ccc);
4: ((c)t).mm = new Integer(999);
```

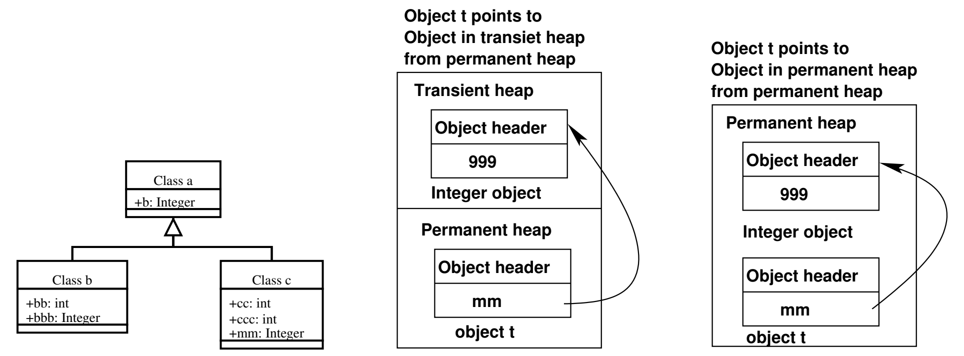
```
1: S.setValue(t);
```

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

Figure 8. Running example used to explain the intra-procedural data-flow analysis

the *instance* size of the `Signal` class, the second element is the allocation program point [§] for the example in Figure 8a. 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.

[§]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) UML class hierarchy for classes `a`, `b` and `c` used in Figure 8b (b) Pointers from permanent heap to transient heap (c) Points-to only in permanent heap

Figure 9. The UML representation of the class hierarchy used in Figure 8 and the points-to problem

★ 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 Figure 8, 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 references are emitted via signals.

★ Step3 analyzes the emission expression for every program point in set PPC. An emitting expression can *only* emit a method local variable or a field. Either Step4 or Step5 are called, depending on whether a field or a local variable is affected, respectively. In our current example a local variable (`t`) in Figure 8b, BB4, line 1, 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 'BB4:line 1', $alloc\_pps = \{\texttt{MCall}:\texttt{BB2}:2, \texttt{MCall}:\texttt{BB3}:3\}$. Looking at Figure 8b, it is clear that object reference `t` 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 guarantees that the object reference does not *escape* the lifetime of the method [17]. [¶]
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 the permanent heap.
Consider the program point in BB3 at line 3. Object `t` is initialized as class `c` (for program point in BB2 line 2, `t` is initialized and analyzed as class `b`), which has three fields (c.f. Figure 9a): 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 in BB3 line 4. Figure 9b 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 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 9c.

[¶]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.

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 from amongst all the different types being initialized. For the running example, t 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 [‖] 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 t in Figure 8b, to escape method MCall, then any of the reference fields of t (e.g., mm) 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 [15] to obtain the program points defining the variable. If the defining program points are new or newarray bytecodes [18], 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 [19] 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 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 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 10.
In Figure 10a, 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 10b. 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

[‖]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

Figure 10. Inadvertent object sharing problem

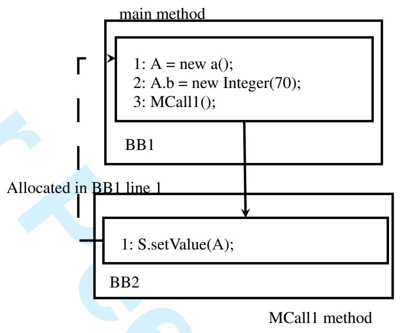


Figure 11. Inter-procedural analysis with fields. `A` and `S` are static fields

(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 10c, 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 permanent heap. Consequently, first a constant 0 will be written in the allocated memory (Figure 10b, line 3) and then this value will be overwritten to 1 (Figure 10b, 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, 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 8b 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. Figure 7) 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 Figure 11 to describe Step4 and Step6 of our data-flow analysis.

★ 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 Java program call-graph in Figure 11 to explain the inter-procedural analysis. The `main` method initializes two objects: `A`, a static field, of type `a` (c.f. Figure 9a) and 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 Figure 11, 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 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 7) 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 *three* types of bytecodes in the JVM specification: (1) the `new` bytecode that allocates an object. (2) The `newarray` bytecode that allocates arrays holding primitives, and (3) `anewarray` bytecode that allocates arrays holding references. This distinction is important since the number of bytes used to represent each of these bytecode differ. 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: Figure 12b, line 3 replacing `new`, line 8 replacing `newarray`, and line 9 replacing `anewarray` bytecodes,

```
Input: maxMem: Maximum memory address
/* defs is the same set as pp in Figure 7 */
Input: defs: The program points to replace
Input: program: The program
let allocPtr ← maxMem;
let header_size ← 2;
foreach (size, dd) ∈ defs do
    let dd ← sortUniqueDescending (dd);
    let M ← load (dd);
    let allocPtr ← allocPtr − (size + header_size);
    if allocPtr < 0 then raise No_mem;
    foreach d ∈ 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 ← getArg (pp);
3      replaceNew (pp,M,arg);
4      adjustConditionals (M,22);
5  else if pp = newarray then
6      let arg ← 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, and anewarray bytecodes

Figure 12. The back-end code generation pseudo-algorithm

respectively. All methods have the same logic, except that some padding (represented as nop bytecodes) is needed in case of the newarray and anewarray bytecodes. This padding is 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 13a and 13b, which show the Java bytecode snippets for the example program in Figure 11 before and after new bytecode replacement, respectively**. Figure 13a 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 and time predictable bytecodes as shown in Figure 13b.

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 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 13c), 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 13b. The second word of the object header is the pointer to the start of the method structure of the class (c.f. Figure 4b). The second word of the object header is initialized in lines 14-21 in Figure 13b.

In case of newarray and anewarray bytecodes, the array layout (shown in Figure 13d) 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 (usually 8 words), which uses spines [20] or handles [14] for

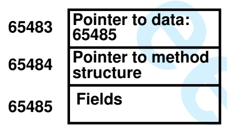
**In both these figures, the line numbers are shown on the right of the program. The numbers adjacent to the bytecodes are the starting bytes for each of the bytecodes as obtained by the javap -c -v command

```
--- 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;
```

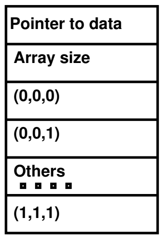
(a) Java bytecode snippet and class constant-pool for example in Figure 11

```
--- 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;
```

(b) Java bytecode snippet and class constant-pool for example in Figure 11 after new bytecode replacement



(c) Java object allocation in permanent heap



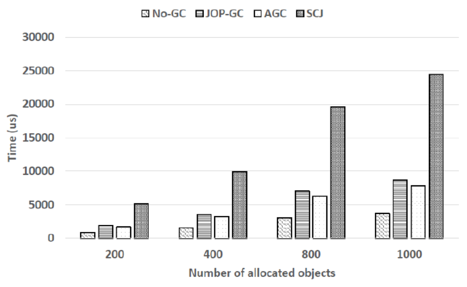
(d) Java (2x2x2) array allocation in permanent heap

Figure 13. The back-end generated code and compacted object representation in heap

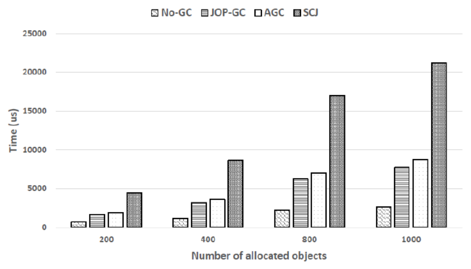
arrays and regular objects, respectively, we also structure our arrays differently. We have decided to place arrays in a row major order and contiguously irrespective of the array dimensions (as in ‘C’) rather than using linked lists for array traversal in order to allow $O(1)$ array accesses.

7. EXPERIMENTAL RESULTS

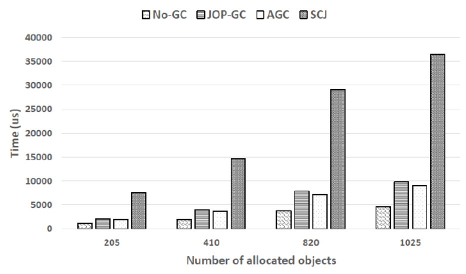
We have carried out two sets of experiments: (a) micro-benchmarks and (b) static WCRT analysis on well established real-time benchmarks to study the efficacy of the proposed memory management scheme compared to: (1) two different real-time garbage collector (RTGC) implementations [14, 21] and (2) The *Safety Critical Java* (SCJ) [22] implementation on the JOP platform. The first RTGC framework is the one proposed in [14], which is a concurrent version of Cheney’s copy collector developed by Baker [23]. In the rest of the section we call it JOP-GC. The second one proposed in [21] is a variation of the real-time collector described in [20], 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, simulating 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.



(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.

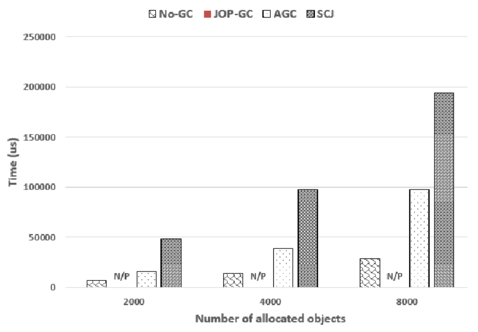
Figure 14. Runtime for micro-benchmarks without GC invocation.

7.1. *Micro benchmarks*

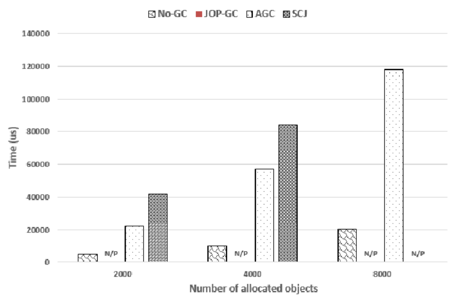
We experiment with three micro-benchmarks: (1) simple object allocation, where we allocate, in a tight loop, 2 objects with 1 word primitive field each. (2) Simple array allocation, where we allocate, in a tight loop, an object with an array type reference field, which is itself initialized to a 20 word 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 are allocated and the second object's reference is set as the array value. The example benchmarks are available from [21].

The runtime comparison between the four approaches for each of the micro-benchmarks is shown in Figures 14 and 15. Figures 14 and 15 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× faster compared to the RTGC based approaches. In many cases (see Figure 15) 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.

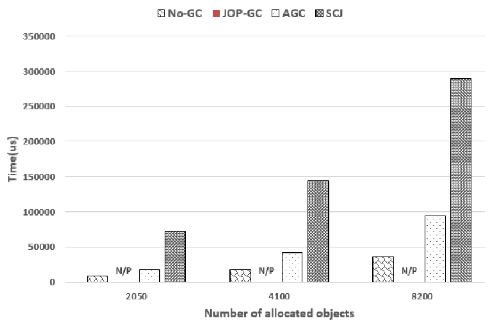
SCJ specification prohibits GC [22] and divides the memory organization into regions such as immortal memory and scoped memory, similar to the permanent and transient heap spaces described in this paper. Given this similarity in the memory organization schemes, we expected to observe memory allocation times, for SCJ, similar to those obtained in the No-GC approach proposed in this paper. Yet, as we see from Figures 14 and 15, the SCJ memory allocation times are far longer compared to all the other memory allocation approaches. In particular, SCJ's memory allocation implementation, in our experiments, is, on average, ≈ 7.3× slower than the proposed approach. Close analysis of the SCJ implementation on the JOP platform indicates that the memory allocation



(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

Figure 15. Runtime for micro-benchmarks with possibility of GC invocation

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)	SCJ (Words)
100	28	2145	2995	2145
200	28	4290	5590	4290
400	28	8580	11180	8580
500	28	10725	13975	10725
1000	28	21450	29950	21450
2000	28	42900	55900	42900
4000	28	85800	111800	85800

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.

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 allocations 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 `newarray` and `anewarray` bytecode execution, array spines need to be initialized for AGC [24, 21], 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 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 `new`, `newarray`, and `anewarray` 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 `new` 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 [††].

SCJ, like the RTGC approaches, allocates memory on each memory allocation bytecode execution. SCJ, unlike the proposed No-GC approach, is unable to reuse memory space allocated in the scoped memory area during the loop iterations. The scoped memory area can only be reused when the task, using the scoped memory area, finishes execution. The No-GC approach greatly benefits from the memory reuse enabled by static analysis proposed in this paper, even though the memory organization of the two approaches is similar.

7.2. *WCRT comparisons*

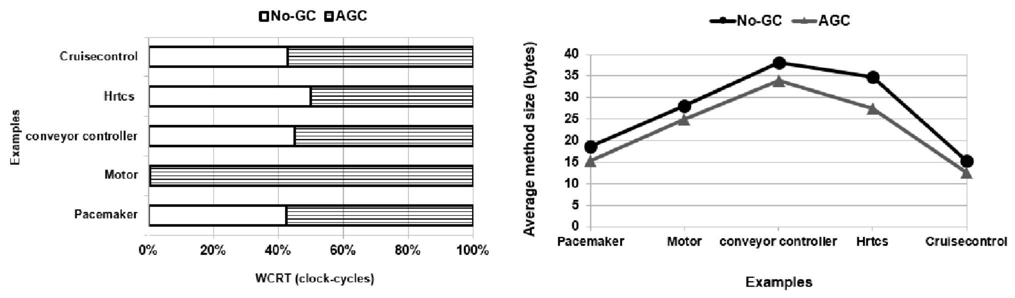
In this section we compare the static WCRT obtained by applying the technique described in Section 3.2 on a set of real-time benchmarks. We chose the benchmarks whose memory allocation requests fit within the allocated heap space without GC invocation. Consequently, we also manually-deleted all code related to GC. This primarily 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 et al. [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 Figures 14 and 15. (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 I) 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` [25] is the cruise speed controller found in cars. `Hrtcs` [26] 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 [8]. `Motor` [27] is a stepper motor control system for a printer. Finally, `Pacemaker` [11], 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 16a.

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 16b 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.

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



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

Figure 16. WCRT and method size comparison: No-GC vs. AGC

8. RELATED WORK AND DISCUSSION

Real-time garbage collectors (RTGCs) such as the ones proposed in [28, 29] and the ones presented in [24], aim to achieve good garbage collection throughput while providing analytic solutions to the worst case execution times of a *single* GC cycle. The major drawback of these approaches is that the equations capturing the GC cycle times 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.

We are not the first to advocate a GC-less approach for hard real-time applications with managed languages, especially Java. There exist previous works like the Safety Critical Java (SCJ) [22] specification and Ravenscar-Java [30] that present a memory organization approach similar to the one presented in this paper. SCJ and Ravenscar-Java both; divide the backing store into multiple parts. A memory region called scoped memory is allocated per real-time task (called a handler in SCJ terminology), which is automatically reclaimed, using pointer reset like us, once the task completes execution. There also exists a permanent memory area similar to the permanent heap advocated in this paper. Any object allocated in the permanent memory area lives throughout the lifetime of the application. One major difference between the presented memory partitioning approach and the memory partitioning in SCJ/Ravenscar-Java is that in the presented approach the same transient heap space (similar to scoped memory in SCJ) can be used by multiple synchronous parallel reactions, whereas every single task gets allocated its own scoped memory in SCJ. This is because, SCJ allows preemption of tasks during program execution, whereas in SystemJ a clock-domain tick is atomic. Furthermore, in this paper we also provide a compile time memory allocation approach, whereas in the SCJ approach, the programmer needs to explicitly and carefully allocate objects to the correct memory areas manually. We believe that our approach 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. The worst case memory analyzer tool [31] developed for SCJ is able to provide the maximum permanent memory and scoped memory sizes needed for each task automatically, like us. 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 applications, because SCJ allows 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.

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 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.

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.

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.

REFERENCES

1. Berry G, Gonthier G. The Esterel synchronous programming language: design, semantics and implementation. *Science of Computer Programming* 1992; **19**(2):87–152.
2. Jagadeesan LJ, Puchol C, Von Olnhausen JE. Safety property verification of esterel programs and applications to telecommunications software. *Computer Aided Verification*, Springer, 1995; 127–140.
3. Malik A, Salcic Z, Roop PS, Girault A. SystemJ: A GALS language for system level design. *Elsevier Journal of Computer Languages, Systems and Structures* December 2010; **36**(4):317–344.
4. Boldt M, Traulsen C, von Hanxleden R. Worst case reaction time analysis of concurrent reactive programs. *Electronic Notes in Theoretical Computer Science* June 2008; **203**(4):65–79.
5. Roop PS, Andalam S, von Hanxleden R, Yuan S, Traulsen C. Tight WCRT analysis of synchronous C programs. *CASES*, 2009; 205–214.
6. Puffitsch W. Design and analysis of a hard real-time garbage collector for a java chip multi-processor. *Concurrency and Computation: Practice and Experience* 2013; **25**(16):2269–2289.
7. Nadeem M, Biglari-Abhari M, Salcic Z. Rjop: a customized java processor for reactive embedded systems. *Proceedings of the 48th Design Automation Conference*, ACM, 2011; 1038–1043.
8. Li Z, Avinash Malik, Salcic ZA. TACO: A scalable framework for timing analysis and code optimization of synchronous programs. *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications, Chongqing, China, August 20-22, 2014*, 2014; 1–8, doi:10.1109/RTCSA.2014.6910556. URL http://dx.doi.org/10.1109/RTCSA.2014.6910556.
9. Hoare C. Communicating sequential processes. *Communication of the ACM* 1978; **21**(8):666–677.
10. Salcic Z, Malik A. GALS-HMP: A heterogeneous multiprocessor for embedded applications. *ACM Trans. Embedded Comput. Syst.* 2013; **12**(1s):58, doi:10.1145/2435227.2435254. URL http://doi.acm.org/10.1145/2435227.2435254.
11. Park H, Avinash Malik, Nadeem M, Salcic ZA. The cardiac pacemaker: Systemj versus safety critical java. *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; 37, doi:10.1145/2661020.2661030. URL http://doi.acm.org/10.1145/2661020.2661030.
12. Schoberl M. JOP: A java optimized processor. *Workshop on Java Technologies for Real-Time and Embedded Systems*, 2003.
13. Behrmann G, David A, Larsen KG. A tutorial on uppaal. *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, Bernardo M, Corradini F (eds.), Springer Verlag, 2004; 200–237. URL http://doc.utwente.nl/51010/.
14. Schoeberl M. JOP: A Java Optimized Processor for Embedded Real-Time Systems. PhD Thesis, Vienna University of Technology 2005. URL http://www.jopdesign.com/thesis/thesis.pdf.
15. Muchnick SS. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.
16. Avinash Malik. Supplementary material. http://homepages.engineering.auckland.ac.nz/~amal029/supplimentary_material.pdf.

17. Choi JD, Gupta M, Serrano M, Sreedhar VC, Midkiff S. Escape analysis for java. *Acm Sigplan Notices* 1999; **34**(10):1–19.
18. Meyer J, Downing T. *Java virtual machine*. O'Reilly & Associates, Inc., 1997.
19. Burke M, Carini P, Choi JD, Hind M. Flow-insensitive interprocedural alias analysis in the presence of pointers. *Languages and Compilers for Parallel Computing*. Springer, 1995; 234–250.
20. Pizlo F, Ziarek L, Maj P, Hosking AL, Blanton E, Vitek J. Schism: fragmentation-tolerant real-time garbage collection. *ACM Sigplan Notices*, vol. 45, ACM, 2010; 146–159.
21. Pathirana I. Aucklandgc: a real-time garbage collector for the java optimized processor. Master's Thesis, University of Auckland 2013.
22. JSR 302: Safety Critical Java Technology. http://jcp.org/en/jsr/detail?id=302 2013.
23. Baker Jr HG. List processing in real time on a serial computer. *Communications of the ACM* 1978; **21**(4):280–294.
24. Pizlo F, Petrank E, Steensgaard B. A study of concurrent real-time garbage collectors. *ACM SIGPLAN Notices*, vol. 43, ACM, 2008; 33–44.
25. Charles A. Representation and analysis of reactive behaviors: A synchronous approach. *Computational Engineering in Systems Applications, CESA*, vol. 96, 1996; 19–29.
26. Park H, Malik A, Salcic Z. Time Square – marriage of real-time and logical-time in GALS and synchronous languages. *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2014*, 2014.
27. Bourke12 T, Sowmya A. Delays in esterel. *SYNCHRON09* 2009; :55.
28. Gestegard-Robertz S, Henriksson R. Time-triggered garbage collection. *Proceedings of the ACM SIGPLAN Langauges, Compilers, and Tools for Embedded Systems* 2003; .
29. Kalibera T, Pizlo F, Hosking AL, Vitek J. Scheduling hard real-time garbage collection. *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, IEEE, 2009; 81–92.
30. Kwon J, Wellings A, King S. Ravenscar-java: a high-integrity profile for real-time java. *Concurrency and Computation: Practice and Experience* 2005; **17**(5-6):681–713.
31. Andersen JL, Todberg M, Dalsgaard AE, Hansen RR. Worst-case memory consumption analysis for scj. *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, ACM, 2013; 2–10.

10. DATA-FLOW ANALYSIS PSEUDO-ALGORITHMS

Figure 17 gives the pseudo-code implementing the data-flow analysis algorithm for compile time memory allocation.

```
Input: program: Program in RTL format
/* pp is the set of program points */
let global $pp \leftarrow \emptyset$;
let $M \leftarrow$ main;
let $M \leftarrow$ ReachableMethods (main) $\cup \{M\}$;
let $size \leftarrow$ sizeof (signal class);
foreach $m \in M$ do
    let pp $\leftarrow$ ($size$, get in $m$, program points initializing signals);
end
Step2 (program,pp,main);
```

(a) Step1:Scan signal initialization

```
Input: program: Program in RTL format
Input: pp: Set of program points
Input: method: Method to scan for signal emissions
let $S \leftarrow$ signalSet (pp);
let $C \leftarrow$ code (method);
let $PPC \leftarrow \emptyset$;
foreach $c \in C$ do
    if emits $(c,S)$ then
        $PPC \leftarrow PPC \cup \{c\}$;
    end
end
Step3 (program,$PPC$,$C$);
foreach $c \in C$ do
    if invokesMethod $(c)$ then
        Step2 (program,$c$,getInvoked ($c$));
    end
end
```

(b) Step2: Scan setValue method calls

```
Input: program: Program in RTL format
Input: $PPC$: Set of program points affected
Input: $C$: code of the method containing $PPC$
foreach $ppc \in PPC$ do
    if affectsFields (getExpr $(ppc)$) then
        let $pp \leftarrow pp \cup$ Step4 (program,$ppc$,getExpr ($ppc$),$C$);
    else if affectsVars (getExpr $(ppc)$) then
        let $pp \leftarrow pp \cup \{$Step5 (program,$ppc$,getExpr ($ppc$),$C$)$\}$;
    else raise error
end
```

(c) Step3: For every affected field/variable get the object allocation program points

```
Input: program: Program in RTL format
Input: $ppc$: Affected program point
Input: $pexpr$: Expression affecting field
Input: $C$: code of the method containing $ppc$
let $f \leftarrow$ fieldOf ($pexr$);
let ($size$,$defs$) $\leftarrow$ Step6 (program,$ppc$,$f$,$C$);
if $size \neq 0$ then return ($size$,$defs$);
foreach $d \in defs$ do
    if !escape $(program,C,d)$ then
        if hasRefFields $(f)$ then
            let $PPC \leftarrow$ getAffectedFields (getFields ($f$));
            Step3 (program,$PPC$,$C$);
        end
    else raise error;
end
let $size \leftarrow$ max (sizeof ($defs$));
return ($size$,$defs$);
```

(d) Step4: Get object allocation program points for affected field

Figure 17. Data-flow analysis pseudo-algorithm

11. THE GRC ANALYSIS ALGORITHM

The algorithm for obtaining *may* and *must* relationships between the GRC nodes is shown in Algorithm 1. The input to this algorithm is an GRC, which is a tuple (V, E) as explained previously. The first part of the algorithm, lines 1-5, groups all the nodes, which are belonging to the same clock-domain. For instance, lines 3-4 initializes a set V_r, which consists of a root node of all clock-domains in GRC. The algorithm then traverses every clock-domain graph, starting from these root nodes via a recursive function collect_cd_nodes (v, E) (line 5). A result is a set of set of GRC nodes V_{cd}. Next *May* and *Must* analysis is performed (lines 6-10). Every node in V_{cd} is given as an input to the function create_node (v, E) (line 9), which traverses the graph backward starting

```
Input: program: Program in RTL format
Input: $ppc$: Affected program point
Input: $pexpr$: Expression affecting field
Input: $C$: code of the method containing $ppc$

let $v \leftarrow$ varOf ($pexr$);
let $defs \leftarrow$ Step7 (program,$ppc$,$v$,$C$);
foreach $d \in defs$ do
    if !escape (program,$C$,$d$) then
        if hasRefFields ($v$) then
            let $PPC \leftarrow$ getAffectedFields
            (getFields ($v$));
            Step3 (program,$PPC$,$C$);
        end
    else raise error;
end
let $size \leftarrow$ max (sizeof ($defs$));
return ($size$,$defs$);
```

(e) Step5: Get object allocation program points for affected variables

```
Input: program: Program in RTL format
Input: $ppc$: Affected program point
Input: $f$: Field affected
Input: $C$: code of the method containing $ppc$

let $ofields \leftarrow$ getAllFields ($C$) $\setminus \{f\}$ ;
let $pcs \leftarrow$ reachDef ($ppc$,$f$);
if $pcs \neq \emptyset$ then
    let $defs \leftarrow \emptyset$;
    foreach $pc \in pcs$ do
        let $vf \leftarrow$ getVarSetting ($f$,$pc$);
        let $tt \leftarrow$ Step7 (program,$pc$,$vf$,$C$);
        if all other $field \in ofields$ have same $tt$
        then
            /* Concat in set $defs$ */
            $defs \leftarrow defs \cup tt$;
        else /* Append a new set in $defs$ */
            $defs \leftarrow defs \cup \{tt\}$;
    end
    return (0,$defs$);
else
    /* Inter-procedural analysis */
    /* Get all callers, including
    from GRC analysis */
    let $Callers \leftarrow$ getCallers ($C$);
    if $Callers = \emptyset$ then raise Not_init $f$;
    let $defs \leftarrow \emptyset$;
    let $sizes \leftarrow \emptyset$;
    foreach $caller \in Callers$ do
        let $defs \leftarrow defs \cup$ Step6
        (program,getCallerPPC ($C$),$f$,$caller$);
        /* Similar to Figure 17d
        lines 4-12, to fill in the
        sizes set. */
    end
    let $size \leftarrow max(sizes)$;
    return ($size$,$defs$);
end
```

(f) Step6: Search object allocation program points for affected field

```
Input: program: Program in RTL format
Input: $ppc$: Affected program point
Input: $v$: Variable affected
Input: $C$: code of the method containing $ppc$

let $pcs \leftarrow$ reachDef ($ppc$,$v$);
if $pcs = \emptyset$ then raise Not_init;
let $defs \leftarrow \emptyset$;
foreach $pc \in pcs$ do
    if getExpr ($pc$) is New or getExpr ($pc$) is
    NewArray then
        $defs \leftarrow defs \cup \{pc\}$;
    else if getExpr ($pc$) is AffectField then
        let $expr \leftarrow$ getExpr ($pc$);
        let (_,$defs$) $\leftarrow$ Step4 (program,$pc$,$expr$,$C$);
    else if getExpr ($pc$) is AffectVar then
        let $v \leftarrow$ varOf (getExpr ($pc$));
        let (_,$defs$) $\leftarrow$ Step7 (program,$pc$,$v$,$C$);
    else
        /* Cannot handle new outside
        current method allocation */
        raise Not_handled
    end
end
/* Make sure that definitions are
not reachable from each other */
if $|defs| > 1$ then
    assert(notReachable ($defs$));
end
return ($defs$);
```

(g) Step7: Search object allocation point(s) for method local affected variable

Figure 17. Data-flow analysis pseudo-algorithm

from the input until it reaches the root node. Function `create_node` first maps the input node to a set of edges E_i using the lambda function (line 14). It is then divided into three parts:

```
   Input: $GRC : (V, E)$
   Output: $mList$: a set of set of $mNode$
   Data: $V_{cd}$: a set of set of clock-domain nodes
   Data: $mNode$ = [$Name$:string,$Must$:$mNode$,$May$:set of $mNode$]
   /* Grouping nodes for each clock-domain */
1  let $V_r = \emptyset \cup V$
2  let $V_{cd} = \emptyset$
3  foreach $v \in V$ do
4      foreach $(v_i, v_j) \in E$ do if $v_j = v$ then $V_r = V_r \backslash \{v\}$
5  foreach $v \in V_r$ do $V_{cd} = V_{cd} \cup$ {collect_cd_nodes($v, E$)}
   /* Perform May and Must analysis */
6  forall the $V \in V_{cd}$ do
7      let $V_{temp} = \emptyset$
8      foreach $v \in V$ do
9          $V_{temp} = V_{temp} \cup$ {create_node($v, E$)}
10     $mList = mList \cup \{V_{temp}\}$
11
12 Function create_node($v, E$):
13 begin
14     let $E_i = \lambda(v)$
15     if $v = ActionNode$ then
16         if $|E_i| = 1$ then
17             let $\{(v_i, v_j)\} = E_i$
18             let $newNode = [Name \leftarrow get\_name(v_j); Must \leftarrow$ create_node($v_i, E$); $May \leftarrow Null]$
19             return $newNode$
20         else if $|E_i| > 1$ then
21             let $V_{temp} = \emptyset$
22             foreach $(v_i, v_j) \in E_i$ do
23                 $V_{temp} = V_{temp} \cup$ {create_node($v_i, E$)}
24             return $[Name \leftarrow get\_name(v_j); Must \leftarrow Null; May \leftarrow V_{temp}]$
25         else if $|E_i| = 0$ then
26             return $[Name \leftarrow get\_name(v_j); Must \leftarrow Null; May \leftarrow Null]$
27     else if $v = JoinNode$ then
28         let $v$ = find_matching_fork($v$)
29         let $\{(v_i, v_j)\} = \lambda(v)$
30         return create_node($v_i, E$)
31     else
32         if $|E_i| = 0$ then
33             return $Null$
34         else if $|E_i| = 1$ then
35             return create_node($v_i, E$)
36
37 Function collect_cd_nodes($v, E$):
38 begin
39     let $V_{temp} = \emptyset$
40     foreach $(v_i, v_j) \in E$ do
41         if $v = v_i$ then
42             $V_{temp} = V_{temp} \cup \{v_j\}$
43             $V_{temp} = V_{temp} \cup$ collect_cd_nodes($v_j, E$)
44     return $V_{temp}$
```

ALGORITHM 1: May and Must analysis

1. If the current node type is an `JDN` node (line 15) and

   (a) the number of element in E_i is 1, then the only parent node v_i *must* be called before v_j. Then a new node of type $mNode$ is created with its field $Must$ initialized to the result of recursive call of `create_note`.

   (b) the number of element in E_i is greater than 1, then any parent nodes $v_i \in E_i$ *may* be called before v_j. A set of $mList$, consisting return results of `create_node` of each parent node $v_i \in E_i$, is assigned to the field May of a newly created $mNode$.

   (c) the number of element in E_i is 0, then a new $mNode$ is created with both its $Must$ and May fields initialized to $Null$.

2. If the current node type is `Join`, this means that there are two or more parallel reactions forked at the `Fork` node, which *must* be found in one of the recursive parents of the current node. Hence, the algorithm must continue the backward traversal from this `Fork` node. During compilation, the compiler has already constructed a Hashtable which maps each `Join` node to its corresponding `Fork` node as shown in line 28.

3. If the current node type is other than `JDN` (line 31) and

   (a) the number of parent node ($|E_i|$) is 0 , then *Null* is returned

   (b) the number of parent node is 1, then a result of `create_node` is returned.