Safety-Critical Java on a Time-Predictable Processor

Stephan E. Korsholm
VIA University College
Horsens, Denmark
sek@via.dk

Martin Schoeberl, Wolfgang Puffitsch
Department of Applied Mathematics and Computer Science
Technical University of Denmark
masca@dtu.dk, wopu@dtu.dk

ABSTRACT

For real-time systems the whole execution stack needs to be time-predictable and analyzable for the worst-case execution time (WCET). This paper presents a time-predictable platform for safety-critical Java. The platform consists of (1) the Patmos processor, which is a time-predictable processor; (2) a C compiler for Patmos with support for WCET analysis; (3) the HVM, which is a Java-to-C compiler; (4) the HVM-SCJ implementation which supports SCJ Level 0, 1, and 2 (for both single and multicore platforms); and (5) a WCET analysis tool.

We show that real-time Java programs translated to C and compiled to a Patmos binary can be analyzed by the AbsInt aiT WCET analysis tool. To the best of our knowledge the presented system is the second WCET analyzable real-time Java system; and the first one on top of a RISC processor.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

Keywords

Safety-Critical Java, hardware locks, synchronization

1. INTRODUCTION

Embedded devices are often used in scenarios where time predictability and real-time behavior is a requirement. In such scenarios it must be possible to guarantee that a given piece of software (e.g., an interrupt handler) will never take more than a certain amount of time to execute. This upper bound to execution time, the worst-case execution time (WCET), is found prior to runtime by analysis tools executed by the software developer. The input to these tools is knowledge about the hardware that will eventually execute the program and the program itself.

Modern day hardware architectures may support features, e.g., caches, speculation, and out-of-order execution, that are designed to achieve a higher average-case execution time. Also for the software part, features, such as garbage collection or just-in-time compilation, are often used to achieve a higher average-case throughput. Unfortunately these features can make the job of finding an accurate WCET harder, or even impossible. For an execution environment to be fully time-predictable, both the hardware and the software parts of the environment must be time-predictable.

JTRES '15, October 07-08, 2015, Paris, France

http://dx.doi.org/10.1145/2822304.2822309.

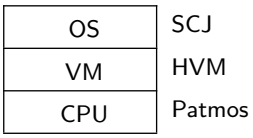

Figure 1: Time-Predictable execution stack for Java.

This paper presents a time-predictable execution environment for real-time and safety-critical Java (SCJ). The execution environment consists of a time-predictable processor and its compiler, a real-time Java-to-C compiler, the runtime for SCJ, and a WCET analysis tool.

The Patmos [31, 35] processor is a time-predictable RISC processor. It comes with a C compiler tool chain for programming the Patmos processor using the C programming language. Additionally, Patmos comes with tools for analyzing C programs and finding the WCETs of selected software components executed by Patmos.

The HVM is a Java-to-C compiler that translates any Java program into an equivalent C program. The HVM-SCJ is an implementation of the SCJ specification [18] hosted by the HVM. SCJ augments the Java programming language with predictable scheduling of concurrently executing schedulables and time-predictable memory management.

This paper describes how the Patmos processor, the HVM, and the HVM-SCJ implementation can be combined to form a WCET analyzable real-time Java system: The HVM Java-to-C compiler translates SCJ based programs into a format that can be executed on the Patmos processor. The Patmos tools for reporting the temporal properties of C code, e.g., WCET analysis, can now be applied to the SCJ programs hosted by the HVM. The combination of these facilities ensures the time-predictable execution of Java on the Patmos hardware platform, thus adding Java as a language alternative.

Figure 1 illustrates the proposed three-layer execution stack for time-predictable Java. It is made up of (1) the hardware platform (Patmos) (2) the Java virtual machine (the HVM) and (3) the OS (SCJ).

The contributions of this paper are:

- A describtion of how to execute the HVM + SCJ software on the Patmos hardware platform.
- Combining the Java-to-C compilation facility of the HVM with Patmos tools for performing WCET analysis of the C code that results from the Java application.
- An evaluation of applying the HVM/SCJ/Patmos tools to time critical software components from well-known benchmarks.

This paper is organized in 8 Sections. Section 2 presents related work. The following three sections describe each of the 3 main components of the execution stack for time-predictable Java programs: Section 3 describes the hardware platform, Section 4 the Java Virtual Machine, and Section 5 the SCJ library implementation making up the software part of the tool chain. Section 6 describes the offline tools used to perform WCET analysis. Section 7 demonstrates how well-known real-time benchmarks are analyzed for their WCET and can be executed using HVM and Patmos. Measurements are reported on the conservatism and efficiency of the tools. These measurements are contrasted to previous results. Section 8 concludes and sums up the contribution and results.

2. RELATED WORK

The Java processor JOP [29] was developed to be a time-predictable processor. It has a simple pipeline where the execution time of individual bytecodes are independent from each other, which simplifies WCET analysis. Furthermore, JOP contains a method cache [28] to simplify cache related WCET analysis. JOP also provides a prototype implementation of SCJ [34]. The distribution of JOP contains the WCET analysis tool WCA [33] that performs WCET analysis at bytecode level. To the best of our knowledge the JOP processor and the included static WCET analysis tools was the first real-time Java system that supports and includes WCET analysis.

In contrast to JOP, we present in this paper a time-predictable real-time Java system that executes on a RISC style processor that executes C code as well. To the best of our knowledge this is then the second real-time Java infrastructure that is supported by a WCET analysis tool.

A first approach to make the HVM time-predictable is the HVMtp project [21]. The interpreter of the HVM has been refactored to be WCET analyzable.

In contrast to HVMtp, we use the compilation mode of HVM, which results in a system with higher performance. Furthermore, model checking for WCET analysis can quickly degrade to explicit path enumeration [16], which is known to not scale for realistic problems [38]. WCET analysis of realistic Java benchmarks on HVMtp has not yet been shown.

The work on PERC Pico [22] was the starting grounds for the early work on Java for real-time systems that later became the Real-Time Specification for Java (RTSJ) [5].

Plsek et al. present one of the first implementations of SCJ on an embedded platform [24]. They provide an implementation of SCJ's Level 0 running on the OVM virtual machine [3]. The OVM is a framework that enables alternate implementations of core VM functionality (e.g., different versions of priority inheritance monitors) in order to build and test VMs with different features. OVM uses an ahead-of-time compiler to translate Java code to C++ and then it uses the GCC compiler to obtain machine code. SCJ's implementation on OVM runs on an FPGA board executing the RTEMS real-time operating system on a LEON3 processor. As LEON3 is supported by the aiT WCET analysis tool, this runtime could be WCET analyzable as the one we present here. The Fiji VM [23] also supports execution of SCJ Level 0 on the LEON3 platform.

Experiments with the Jamaica ahead-of-time compiler lead to programs where the aiT analyzer was not able find loop bounds. Therefore, additional data-flow analysis at bytecode level can provide those loop bounds [17]. In our current implementation we use manual annotations for loop bounds. Therefore, such a data-flow analysis at bytecode level would be a useful enhancement for our presented platform.

3. PATMOS

Patmos is a dual-issue RISC processor designed to be a time-predictable platform to simplify WCET analysis [35]. Patmos was developed within the T-CREST project [30], which developed a time-predictable chip-multiprocessor with a real-time network-on-chip and a real-time memory controller. Furthermore, the LLVM compiler and the aiT WCET analysis tool [11] were adapted to Patmos.

3.1 Pipeline and Instructions

Patmos is an in-order pipeline. To gain time-predictable performance compared to out-of-order pipelines, Patmos can execute up to two instructions per clock cycle. Patmos has a fully predicated instruction set. Those predicates help with if-conversion, an optimization to avoid branches. Furthermore, predicates also support a similar technique that is used to generate single-path code [25].

Another speciality of Patmos are typed load and store instructions. The types are used to e.g., redirect memory accesses to a special cache, to bypass the cache, or to access local memory (scratchpad memory). Therefore, a WCET analysis tool has more information on which memory area a load or store instruction will access without needing to know the concrete memory address.

3.2 Caches

A dominant factor of execution time and also WCET are cache misses. A lot of research effort has been spent to analyze instruction caches and data caches for their contribution to the WCET. With Patmos we developed special caches with the intention to make the WCET analysis simpler and the WCET bound lower.

For instructions Patmos contains the so called method cache [6] that caches complete functions. The cache is called method cache as it has been originally implemented for the Java processor JOP to cache whole methods [28]. The method cache may be filled on a function call or a function return. Fetching of all other instructions are guaranteed cache hits. Therefore, WCET analysis has only to consider function call and return instructions.

One common and efficient compiler optimization is function inlining. This optimization, and original large functions, can result in functions that are too large for the method cache. Therefore, the Patmos compiler contains a pass to split functions [12]. Function splitting is also used as an optimization to avoid loading unused code into the method

cache. Function splitting is supported by low overhead instructions in Patmos.

The method cache can be integrated into WCET analysis by a scope based analysis [13]. This analysis searches large scopes in the global control flow graphs with functions that fit together into the method cache. Within this program scope these functions can only miss once and all other calls are hits. This information is added as ILP constraints to the ILP for the WCET.

Data caches usually cache data that belong to different data areas, such as static data, heap allocated data, and stack allocated data. For standard cache analysis the address of data accesses needs to be known statically. Knowing the address for static data is easy, for stack allocated data possible, and for heap allocated data impossible.

To avoid intermixing of these data types Patmos contains a stack cache [1]. The stack is, especially in a Java runtime, heavily used and therefore deserves special support in the processor. This stack cache is supported by the compiler that generates instructions for stack frame manipulation. A *reserve* instruction reserves space in the stack cache and might spill data to the main memory to make space. A *free* instruction just marks space in the stack cache as unused. An *ensure* instruction ensures that the callers stack frame is in the stack cache after a return from the callee. This instruction might fill former spilled data from main memory to the stack cache. Only the *reserve* and *ensure* instructions might access main memory. All other accesses to stack allocated data are guaranteed hits. This mechanism simplifies the WCET analysis for stack allocated data [19].

3.3 Composable WCET Analysis for Patmos

In general, the architecture of Patmos is designed that there are no timing anomalies and intended to support composable WCET analysis. That means that no two instructions timing depend on each other. Cache misses happen all in the same (memory) pipeline stage. Therefore, only a single instruction at any clock cycle might trigger a cache miss. These features allow to decouple the pipeline analysis from the analyses of different caches.

4. HVM-AOT

Java applications hosted by the HVM can be compiled into a mix of interpreted and AOT compiled code. The interpreted part consists of a sequence of Java bytecodes that is interpreted by a standard interpreter loop. The Java bytecodes are embedded into the C program as a C array of bytes. The interpreter loop is basically a large switch statement enclosed in a loop. The AOT compiled part is a sequence of auto-generated C functions, one for each original Java method. The name of the C function is derived from the Java method name, the name of the enclosing Java class and the name of the Java package containing the enclosing class. Control can flow from interpreted to AOT compiled parts and back again. Both methods of execution are supported since they both have their strengths and weaknesses: interpretation yields slower execution but takes up less code memory resources, AOT compiled code yields faster execution, but requires more code memory resources. The developer can configure which methods are compiled and which methods are interpreted. The resulting C source (containing the interpreter) is compiled by a C cross compiler and linked with the HVM runtime system to produce the final executable for the target in question. If the Java source contains native methods, an implementation of those methods in C must be provided as well.

The HVM tools are careful to generate portable C code that can be compiled readily by any compliant C compiler.

In this paper we focus on the Java-to-C translation facility of the HVM. A previous paper has looked at the interpreter and how the interpreter can be made time-predictable as well [21].

To reduce the code memory footprint of the resulting application, the HVM tools perform an analysis of the application to find a conservative, but tight, estimate of the set of methods and classes that may be executed at run time. This set is called the dependency extent of the main entry point. The HVM tools support that Java methods can be excluded from the dependency extent and implemented in native C in a similar manner to normal native methods.

The dependency extent is calculated by visiting all possible traces of the program execution starting from the main method of the program. For if-statements the dependency extents of all branches are conservatively added. For virtual method invocations a similar simple choice cannot be made. Indeed, for a virtual method invocation the question of which method(s) could be the target of the invocation, and thus needs to be included in the extent, can only be answered if it is known which classes might have been instantiated along all possible execution traces leading up to the method invocation. The analysis tool keeps track of which classes might have been instantiated. If analysis arrives at a virtual method invocation and the set of possibly instantiated classes is larger than or different from when the method invocation was previously visited, the analysis continues until a fixed point is reached. This method excludes dynamic classloading, which is consequently not supported by the HVM runtime.

```
class Polygon {
    abstract int area();
}
class Square extends Polygon { ... }

class Rectangle extends Square { ... }

class Circle extends Polygon { ... }

...

ArrayList<Polygon> figures = new ArrayList<Polygon>();
figures.add(new Square(2));
figures.add(new Rectangle(2, 3));
figures.add(new Circle(3));

int sum = 0;
for (Polygon polygon : figures) {
   sum += polygon.area();
}
```

Figure 2: Translating virtual method invocations (Java Source).

Figure 2 shows a Java source method with a virtual method invocation (`sum += polygon.area()`). The target method (`area()`) can be any of three methods. The HVM will translate this into the C code included in Figure 3, i.e., a switch

statement. In each case the target method is called through a direct function invocation.

```
switch (classIndex) {
   case 18:
      rval_m_85 = test_Circle_area(sp, i_val3);
      break;
   case 30:
      rval_m_85 = test_Rectangle_area(sp, i_val3);
      break;
   case 5:
      rval_m_85 = test_Square_area(sp, i_val3);
      break;
}
```

Figure 3: Translating virtual method invocations (C source).

The JVM is a stack-based virtual machine. Each Java method uses a stack of a known size to perform all calculations done by the byte codes making up the application. Figure 4 is an example of a piece of code adding two numbers.

```
ICONST_1
ISTORE_0
ICONST_2
ISTORE_1
ILOAD_0
ILOAD_1
IADD
ISTORE_0
```

Figure 4: Adding two numbers in bytecode.

To avoid simulating each stack access - which would be inefficient - the HVM assigns a C variable to each stack cell. The resulting generated code is listed in Figure 5.

```
LV2 = 1;
LV0 = LV2;
LV2 = 2;
LV1 = LV2;
LV2 = LV0;
LV3 = LV1;
LV3 = LV2 + LV3;
LV0 = LV3;
```

Figure 5: Adding two numbers in C.

It may seem inefficient to use so many variables for such a simple calculation, but fortunately the C compiler that eventually translates this into machine code will be able to optimize this into a very efficient format. Enabling the proper optimization levels is important when using a C compiler to generate the final executable.

Loops in Java sources are implemented using the `goto` Java bytecode. This gets translated into C `goto` statements in the generated C source code.

Using these and other techniques the HVM tools translate the dependency of the Java main method into portable C code. This code will then in turn be analyzed by the Patmos WCET analysis tools as described in Section 6.

4.1 Porting to Patmos

The HVM has previously been ported to a variety of platforms, both 8/16/32 and 64 bit platforms running Linux/Windows/Cygwin and bare-bone platforms. The major challenge with the Patmos platform was that it does not allow unaligned memory access. In its default configuration the HVM generates code that may result in unaligned access to heap memory locations. This is because object headers and object fields are accessed by casting them to byte arrays and fields are packed tightly in the objects in order to save space. When porting the HVM to the Lego NXT platform, which runs on an ARM7 processor that also does not allow unaligned memory access, another mode of object layout and access has been introduced. In this mode the HVM generates a C struct for each Java class, where the struct members are the Java object fields. Figure 6 shows an example of a Java class `Sub` and Figure 7 shows the C struct being generated from it.

```
class Super {
   int x;
}

class Sub extends Super {
   int a, b, c;
}
```

Figure 6: An example Java class.

```
typedef struct PACKED _test_TestPutGetField_Sub_c {
    Object header;
    uint32 x_f;
    uint32 a_f;
    uint32 b_f;
    uint32 c_f;
} test_TestPutGetField_Sub_c;
```

Figure 7: Java class to C struct mapping.

Now AOT compiled code will access field members by casting the object in the heap to the struct type. E.g a `putfield` bytecode to the field `x` in an instance of class `Sub` becomes:

```
(struct _test_TestPutGetField_Sub_c *)(cobj)
      -> x_f = lsb_int32;
```

This is done in a similar manner in other Java-to-C compilers (e.g., KESO [37]). When using the Patmos C compiler the field access no longer crosses alignment boundaries if the `PACKED` macro is left undefined.

5. HVM-SCJ

The Java programming language has a built-in thread concept to support concurrent execution of Java methods. Yet, the scheduling model is not specified in sufficient detail to ensure an acceptable degree of time-predictability of standard Java execution environments. Furthermore the garbage collection facilities specified for standard Java can make it hard to find a sufficiently tight WCET bound for software that

accesses heap memory, since garbage collecting may be activated at any allocation, and take an amount of time that is hard to predict. The latter issue with garbage collection has been solved in Java execution environments such as Perc [4] and JamaicaVM [2] which offer real-time garbage collection. The former issue has been recognized and attacked by defining new profiles for Java such as the Real-Time Specification for Java (RTSJ) [5] and safety-critical Java (SCJ). Both of these profiles define detailed scheduling policies and put restrictions on memory management by offering scoped memory allocation. Memory scopes are a compromise between the `malloc/free` type of manual memory allocation offered by e.g., the C programming language and full automatic memory allocation which is offered by standard Java execution environments. The advantage of memory scopes is that they support a more structured approach to memory allocation than `malloc/free` while still being time-predictable as scopes are allocated and deallocated in a simple time-predictable manner.

5.1 Profiles of the SCJ Specification

The SCJ specification defines three profiles, which are known as Level 0, Level 1 and Level 2 respectively. Level 1 and Level 2 support multicore. The HVM-SCJ implementation supports all three levels:

Level 0 Here a sequence of missions is executed. A mission consists of periodic handlers only, and in the active phase they are scheduled statically by a cyclic executive.

Level 1 Here a sequence of missions is executed. However, a mission may include aperiodic handlers, and in the active phase fixed priority preemptive scheduling is used. Thus interrupt driven handlers are admitted. The preemptive scheduling means that a priority ceiling protocol has to be used for objects shared among handlers.

Level 2 This allows missions to be nested, so they can be run concurrently. Also, at this level, real-time threads are admitted.

Level 0 and level 1 were implemented first [36], because these levels target applications running on resource constrained embedded platforms. As a step towards supporting more resourceful platforms, level 2 has recently been added [39]. This introduces the following concepts,

- Managed threads, scheduled by the priority scheduler
- Wait, notify, and notifyAll for managed schedulable objects
- Nested mission sequencers

Finally multicore support has been added, which is described in another JTRES'15 paper.

5.2 SCJ Memory Management

Since time- and space-predictable approaches to dynamic memory allocation is still considered with some caution by standards for safety-critical applications, the SCJ uses a restricted version of the RTSJ scoped memory concept. Objects are allocated in a memory area with a defined lifetime.

The *immortal memory* is used for allocating objects that live for the entire lifetime of the allocation. The *mission memory* is used for objects that live for the duration of a single mission only. Each schedulable object, e.g., a thread (level 2) or a periodic handler (level 0) has its own *private memory*—this area gets cleared after each invocation of the application logic of the schedulable objects. E.g., the private memory area of a periodic handler is reused for each invocation of the handler. Temporary private memory areas can always be allocated and entered for a period of time and afterwards be deallocated in one single operation. Each of these areas are also referred to as scopes. Allocation and deallocation of memory scopes is a time-predictable operation. Still, using scopes has some of the same disadvantages as other manual memory management strategies like the `malloc/free` strategy for the C programming language. An important area of research within real-time Java deals with tools aiding the developer in avoiding memory errors and ensuring error free scope allocation and deallocation

The HVM-SCJ implementation offers the OS functionality of the architecture illustrated in Figure 1 through its well defined scheduling policies at all levels and through its time-predictable memory allocation facilities.

6. WCET ANALYSIS

WCET analysis for Patmos is provided by a combination of three tools: (1) the LLVM compiler adapted for Patmos [27], (2) the platin tool, and (3) the industry standard static WCET analysis tool aiT [11].

6.1 Compiling for Patmos

Within the T-CREST project LLVM [20] has been adapted for Patmos [26]. The compiler supports all special features of Patmos, i.e., it emits instructions for the stack cache and optimizes programs for the usage of the method cache.

Furthermore, the Patmos compiler can optimize for the WCET by getting feedback from the WCET analysis tool. The compiler also supports `_Pragma` based flow facts as they have been defined for the WCC compiler [7].

The compiler solves the problem of transforming flow facts during optimization by using control-flow relation graphs for the WCET analysis at bitcode and machine code level [14].

6.2 WCET Analysis with aiT

For WCET analysis we use aiT [11], the WCET analysis tool from AbsInt. aiT produces safe upper bounds for the WCET of non-interrupted tasks. aiT supports several processors used in embedded real-time systems. Within the T-CREST project aiT has been extended to support the Patmos instructions set, with support for VLIW architectures, fully predicated instruction set, and analysis for the stack cache and the method cache.

aiT takes as input binary executable and additional information in an AIS file. The additional information in the AIS file contains e.g., memory access times, configuration for caches, and flow facts that might not be detected automatically by the tool.

aiT reconstructs from the binary the control flow graph and performs loop bound analysis and value analysis for to determine approximations for values in registers and memory. These values are used for loop bounds and to determine addresses for the data cache analysis.

Analysis of basic block execution time uses abstract interpretation and simulation of the processor pipeline. Those results at basic block level are then used in search for the longest path with implicit path enumeration. This path is then a safe upper bound of the WCET.

6.3 Tool Integration with platin

`platin` is intended to be a swiss army knife for compiler and analysis integration. The work on `platin` started with an approach to an open timing analysis platform [15]. It is argued that research in the WCET analysis community needs more integration of open-source tools. The work started with an example integration of the LLVM compiler [20], the open source processor LEON [8], the SWEET tool (SWEdish Execution Time tool) [9], and the WCET analysis tool aiT [11]. SWEET was used for program analysis (e.g., generating flow facts) and aiT was used for the WCET analysis, i.e., calculation of the WCET bound. Later, within the T-CREST project, the work in the compiler shifted the focus towards the open-source processor Patmos.

`platin` is a collection of tools to exchange information between the compiler and WCET analysis tools. `platin` is bundled with the Patmos compiler and tightly integrated with it.

For exploration, `platin` supports extraction if flow facts from test runs with the Patmos simulator. This feature comes handy when the research work is on a different topic than generating exact flow facts, e.g., cache analysis.

`platin` also includes a WCET analysis backend using standard implicit path enumeration techniques by translating the WCET analysis problem to an integer linear program, which is then solved with `lp_solve`. Within this WCET tool, the scope based analysis of the method cache of Patmos is [13] and analysis of the stack cache [19] have been integrated.

However, in the evaluation section we use the industry strength aiT tool as it has support for data cache analysis.

7. EVALUATION

To evaluate the analyzability of the HVM on Patmos, we use a set of Java benchmarks that are based on the Mälardalen WCET benchmarks [10]. Additionally, we use two benchmarks from the JemBench suite [32] that are derived from industrial applications. We use the ahead-of-time compilation mode of HVM and compile the code with LLVM for Patmos. For the WCET analysis, we use the WCET analysis tool aiT by AbsInt [30]. We configure the WCET analysis tool to assume a 4 KB method cache with eight blocks, a 2 KB direct-mapped data cache, and a memory access latency of 21 cycles for a burst of 4 words. The memory access time corresponds to the memory access time of Patmos on its default target platform, the Altera DE2-115 development board.

7.1 Annotations

To bound the execution time of a program, it is necessary to bound the number of iterations for all loops. While a WCET analysis tool often can find loop bounds and other flow facts by analyzing the program, this is not always possible. In these cases, annotations have to be provided, either in source code or in a separate file. On the one hand, loops in the application must be bounded; on the other hand, also loops in the HVM-internal code must be bounded.

Ideally, annotations present in the Java source code should be propagated automatically to the generated C code and the WCET analysis tool. The C code generated by the HVM implements loops through `goto` statements rather than `for`- or `while`-loops. However, the loop bound annotations supported by the C compiler (e.g., `_Pragma("loopbound min 2 max 99")`) can only be applied to loops and not to `goto` statements. Extending the C compiler to support more general flow-fact annotations is future work. Consequently, the annotations provided in the source code of the benchmarks currently cannot be propagated automatically.

As automatic propagation of annotations is not possible, we annotated loop bounds where necessary with the help of aiT. As the Java compiler and the HVM retain the structure of the program during compilation, the program structure at the C level closely matches the structure at the Java level. Unfortunately, optimizations by the C compiler sometimes modify the program structure. In particular inlining can make it difficult to correctly match loops at the Java level and the binary level. However, after disabling inlining in the C compiler, entering annotations corresponding to loops in the benchmark code was straightforward.

Of the HVM-internal loops, only one loop in function `addStackElement` could not be bounded automatically for the benchmarks used in the evaluation. However, the necessary information to specify the respective loop bound is present in a generated header file `methods.h` that contain several attributes of the generated application code. For the function `addStackElement` the `NUMBEROFMETHODS` was the correct loop bound. The HVM-AOT compiler also generates another header file `tinfo.h` with additional loop bounds:

- `MAX_APP_STACK` This constant holds the max stack depth (in frames) of the application. This is only accurate for non-recursive applications. Can be used to bound exception handling
- `MAX_CLASS_HEIRARCHY` This constant holds the depth of the deepest class heirarchy.
- `MAX_LOOKUPTABLE_SWITCH_SIZE` This constant holds the size of the biggest lookup table (for the `lookupswitch` and `tableswitch` bytecodes)
- `MAX_INVOKE_TABLE_SIZE` A loop is used to handle the invokevirtual bytecode in the interpreter. This constant is an upper bound for that loop

These additional loop bounds may be relevant for other applications than the benchmarks used here.

Apart from loops, the WCET analysis tool also needs annotations for indirect jumps that may occur for example for `switch` statements. Annotations for these flow facts are generated by `platin`, such that no manual inspection of the code is necessary.

7.2 Exceptions

On the one hand, exceptions may contribute to the execution time and as such should be included in the WCET calculation. On the other hand, a safety-critical application should be free from exceptions to operate correctly. To cater to both points of view, we calculate each WCET under two distinct sets of assumptions: first that the code may throw an exception and second, that it won't. In many safety-critical applications, static analysis of the software proves

Benchmark	Function	WCET Analysis		Measurements	
		Cycles with Exceptions	Cycles without Exceptions	Cycles	Pessimism
BinarySearch	binarySearch	36505	1212	711	1.70
BubbleSort	bubbleSort	1628409	1592461	400555	3.98
DCT	fdct	50939	13957	8011	1.74
ExpIntegral	expint	1338987	1338987	6578	203.56
Fibonacci	fib	898	898	306	2.93
InsertionSort	sort	49891	13743	3404	4.04
JanneComplex	complex	2097	2097	406	5.17
MatrixCount	count	55555	20223	8168	2.48
MatrixMult	multiplyTest	1764128	1728739	499616	3.46
NestedSearch	foo	123402	88115	22031	4.00
PetriNet	run	164659	129427	100014	1.29
Quicksort	sort	564964	141521	6061	23.35
SelectSmallest	select	314030	67178	3931	17.09
SLE	ludcmp	89363	54140	5985	9.05
Kfl	Mast.loop	529940	67894	15657	4.34
Lift	loop	210311	18309	8768	2.09

Table 1: Evaluation results

that exceptions will not be thrown. When it is known that exception handling will not occur, the WCET bounds can be much tighter. Doing so also gives us an insight into the worst-case overhead for exception handling.

7.3 Results

Table 1 shows the results of the evaluation. The columns under the heading "WCET Analysis" report the results of the WCET analysis, while the columns under the heading "Measurements" display the measurement results. Aside from showing worst-case cycle counts, the results demonstrate that it is feasible to calculate WCETs for the code generated by the HVM.

The column labeled "Cycles with Excexptions" shows the WCET including exception handling, while the column labeled "Cycles without Exceptions" shows the WCET under the assumption that exception handling code infeasible. For most benchmarks, exception handling contributes around 35000 cycles or does not contribute at all. The outliers here are the Quicksort, SelectSmallest, Kfl, and Lift benchmarks, where exception handling contributes several hundred thousand cycles.

WCET analysis results for the same benchmarks are also available for JOP [33]. Due to the different hardware setup (in particular memory access times), we refrain from doing a direct comparison. However, the results for both platforms are consistent in the sense that relative times between slow and fast benchmarks are comparable.

The measurement results in Table 1 were obtained with the cycle-accurate simulator for Patmos. The pessimism ratio compares the measurement results with the WCET computed under the assumption that there are no exceptions. The pessimism in our setup is in general higher than the pessimism reported for JOP [33]. The high pessimism for QuickSort and SelectSmallest is also found in the results for JOP, where a pessimism factor of around 10 is reported for these benchmarks. The extremely high pessimism for the ExpIntegral benchmark is caused by a software division routine, which in measurements performs much faster than has to be assumed for the worst case.

8. CONCLUSION

Real-Time systems need a complete time-predictable platform, starting from the processor, to the compiler, the runtime system, and the application code. In this paper we presented a time-predictable processor that is supported by the WCET analysis tool aiT. On top of this processor we provide the real-time Java virtual machine HVM that supports safety-critical Java. On this combination we are able to compile and analyze for the WCET standard WCET benchmarks. This solution is the first real-time Java system on a RISC type processor that is WCET analyzable.

Acknowledgment

We would like to thank the Patmos compiler team: Florian Brandner, Stefan Hepp, Alexander Jordan, and Daniel Prokesch for providing the LLVM adaption for Patmos and the integration with the aiT WCET analysis tool. We are especially thankful to Benedikt Huber for doing it again (after WCA for JOP): implementing the WCET analysis tool platin from scratch for the T-CREST project. We would like to thank Christoph Cullmann and Gernot Gebhard for the adaption of aiT for Patmos and thank AbsInt for providing us their WCET analysis tool.

The work presented in this paper was partially funded by the Danish Council for Independent Research | Technology and Production Sciences under the project RTEMP, contract no. 12-127600.

Source Access

The work described in this paper is available in open source and both projects are hosted at GitHub. The Patmos processor and the adapted LLVM compiler are available at `https://github.com/t-crest`. The HVM and the SCJ implementation are available at `https://github.com/scj-devel`.

9. REFERENCES

[1] S. Abbaspour, F. Brandner, and M. Schoeberl. A time-predictable stack cache. In *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

[2] aicas. http://www.aicas.com/jamaica.html. Visited June 2012.

[3] A. Armbruster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek. A real-time Java virtual machine with applications in avionics. *Trans. on Embedded Computing Sys.*, 7(1):1–49, 2007.

[4] Atego. Aonix Perc Pico. Available at: http://www.atego.com/products/aonix-perc-pico/.

[5] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[6] P. Degasperi, S. Hepp, W. Puffitsch, and M. Schoeberl. A method cache for Patmos. In *Proceedings of the 17th IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2014)*, pages 100–108, Reno, Nevada, USA, June 2014. IEEE.

[7] H. Falk and P. Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems*, pages 1–50, 2010.

[8] J. Gaisler. A portable and fault-tolerant microprocessor based on the SPARC v8 architecture. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, page 409, Washington, DC, USA, 2002. IEEE Computer Society.

[9] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Real-Time Systems Symposium (RTSS 2006), IEEE International*, volume 0, pages 57–66, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[10] T. Harmon, M. Schoeberl, R. Kirner, and R. Klefstad. A modular worst-case execution time analysis tool for Java processors. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, pages 47–57, St. Louis, MO, United States, April 2008. IEEE Computer Society.

[11] R. Heckmann and C. Ferdinand. Worst-case execution time prediction by static program analysis. Technical report, AbsInt Angewandte Informatik GmbH. [Online, last accessed November 2013].

[12] S. Hepp and F. Brandner. Splitting functions into single-entry regions. In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '14, pages 17:1–17:10, New York, NY, USA, 2014. ACM.

[13] B. Huber, S. Hepp, and M. Schoeberl. Scope-based method cache analysis. In *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, pages 73–82, Madrid, Spain, July 2014.

[14] B. Huber, D. Prokesch, and P. Puschner. Combined WCET analysis of bitcode and machine code using control-flow relation graphs. In *Proceedings of the 14th ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems (LCTES 2013)*, pages 163–172. The Association for Computing Machinery, 2013.

[15] B. Huber, W. Puffitsch, and P. Puschner. Towards an open timing analysis platform. In *Proceedings of the 11th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 6–15, 2011. talk: 11th International Workshop on Worst-Case Execution Time Analysis, Porto; 2011-07-05.

[16] B. Huber and M. Schoeberl. Comparison of ILP and model checking based WCET analysis. Technical Report 72/2008, Institute of Computer Engineering, Vienna University of Technology, December 2008.

[17] J. J. Hunt, I. Tonin, and F. Siebert. Using global data flow analysis on bytecode to aid worst case execution time analysis for real-time java programs. In G. Bollella and C. D. Locke, editors, *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems, (JTRES 2008)*, volume 343 of *ACM International Conference Proceeding Series*, pages 97–105. ACM, 2008.

[18] Java Expert Group. Java specification request JSR 302: Safety critical java technology. Available at http://jcp.org/en/jsr/detail?id=302.

[19] A. Jordan, F. Brandner, and M. Schoeberl. Static analysis of worst-case stack cache behavior. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems (RTNS 2013)*, pages 55–64, New York, NY, USA, 2013. ACM.

[20] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–88. IEEE Computer Society, 2004.

[21] K. S. Luckow, B. Thomsen, and S. E. Korsholm. Hvmtp: A time predictable and portable Java virtual machine for hard real-time embedded systems. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages 107:107–107:116, New York, NY, USA, 2014. ACM.

[22] K. Nilsen and S. Lee. Perc real-time api (draft 1.3). newmonics, July 1998.

[23] F. Pizlo, L. Ziarek, and J. Vitek. Real time Java on resource-constrained platforms with Fiji VM. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2009)*, pages 110–119, New York, NY, USA, 2009. ACM.

[24] A. Plsek, L. Zhao, V. H. Sahin, D. Tang, T. Kalibera, and J. Vitek. Developing safety critical Java applications with oSCJ/L0. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010)*, pages 95–101, New York, NY, USA, 2010. ACM.

[25] D. Prokesch, S. Hepp, and P. Puschner. A generator for time-predictable code. In *Proceedings of the 17th IEEE Symposium on Real-time Distributed Computing (ISORC 2015)*, Aukland, New Zealand, April 2015. IEEE.

[26] P. Puschner, R. Kirner, B. Huber, and D. Prokesch. Compiling for time predictability. In F. Ortmeier and P. Daniel, editors, *Computer Safety, Reliability, and Security*, volume 7613 of *Lecture Notes in Computer Science*, pages 382–391. Springer Berlin / Heidelberg, 2012.

[27] P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, and G. Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In *9th Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS 2013)*, pages 33–40, 2013.

[28] M. Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[29] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[30] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, (0):accepted for publication, 2015.

[31] M. Schoeberl, F. Brandner, S. Hepp, W. Puffitsch, and D. Prokesch. Patmos reference handbook. Technical report, 2014.

[32] M. Schoeberl, T. B. Preusser, and S. Uhrig. The embedded Java benchmark suite JemBench. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010)*, pages 120–127, New York, NY, USA, August 2010. ACM.

[33] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

[34] M. Schoeberl and J. R. Rios. Safety-critical Java on a Java processor. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2012)*, pages 54–61, Copenhagen, DK, October 2012. ACM.

[35] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.

[36] H. Søndergaard, S. E. Korsholm, and A. P. Ravn. Safety-Critical Java for low-end embedded platforms. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 44–53, New York, NY, USA, 2012. ACM.

[37] I. Thomm, M. Stilkerich, C. Wawersich, and W. Schröder-Preikschat. KESO: an open-source multi-JVM for deeply embedded systems. In *JTRES'10*, pages 109–119. ACM, 2010.

[38] R. von Hanxleden, N. Holsti, B. Lisper, E. Ploedereder, R. Wilhelm, A. Bonenfant, H. Casse, S. Bünte, W. Fellger, S. Gepperth, J. Gustafsson, B. Huber, N. M. Islam, D. Kästner, R. Kirner, L. Kovacs, F. Krause, M. de Michiel, M. C. Olesen, A. Prantl, W. Puffitsch, C. Rochange, M. Schoeberl, S. Wegener, M. Zolda, and J. Zwirchmayr. WCET tool challenge 2011: Report. In *Proceedings of the 11th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Porto, Portugal, July 2011.

[39] S. Zhao. Implementing level 2 of Safety-Critical Java, 2014.

Build Instructions

As all the technologies presented here are open source we provide build and run instructions for the examples in the paper. Building Patmos and the compiler are briefly described at `https://github.com/t-crest/patmos` and in more detail in the Patmos reference handbook [31] in Chapter 6. Here we describe how to run the JemBench benchmarks using the HVM-AOT and the Patmos simulator.

1. Download the JemBench benchmark suite from `http://sourceforge.net/projects/jembench/`
2. In the JemBench main entry point in `jembench.Main` add a line setting the `args` parameter to null
3. Disable the parallel and stream benchmarks
4. Use Git to clone the icecaptools, icecaptoolstest, icecapvm and icecapSDK modules from github (`https://github.com/scj-devel/hvm-scj`)
5. Import them into Eclipse as existing projects
6. Open the file `hvm.properties` and fix the properties to match your environment. This file is used as input to the compilation process and points the HVM-AOT compiler to its input. Comments have been left in the official version to point to the JemBench main entry point
7. Run the class `main.CompilationManager` as a normal Java application from inside eclipse. This class is located in the icecaptoolstest project
8. In a prompt (e.g. xterm or cygwin) go into the folder `.../hvm-scj/icecapvm/src` (the path must be adapted)
9. Execute the command:

```
cp ../../icecaptoolstest/*.[ch] .
```

10. Compile with (Makefile)

```
all:
    patmos-clang -target patmos-unknown-unknown-elf \
    -mpatmos-method-cache-size=0x1000 \
    -mpatmos-preferred-subfunction-size=0 \
    -mpatmos-stack-base=0x200000 \
    -mpatmos-shadow-stack-base=0x1f8000 \
    -DPACKED= -Wall -pedantic -O3 -DPC32 \
    -DPRINTFSUPPORT -DUSEGETTIMEOFDAY \
    -DJAVA_HEAP_SIZE=1310720 \
    classes.c icecapvm.c methodinterpreter.c methods.c
    gc.c natives_allOS.c natives_i86.c rom_heap.c \
    allocation_point.c rom_access.c native_scj.c \
    print.c -lrt
```

11. Execute the command: `pasim a.out`