Just-in-Time Compilation-Inspired Methodology for Parallelization of Compute Intensive Java Code

GHULAM MUSTAFA*, WAQAR MAHMOOD**, AND MUHAMMAD USMAN GHANI*

RECEIVED ON 02.09.2015 ACCEPTED ON 14.12.2015

ABSTRACT

Compute intensive programs generally consume significant fraction of execution time in a small amount of repetitive code. Such repetitive code is commonly known as hotspot code. We observed that compute intensive hotspots often possess exploitable loop level parallelism. A JIT (Just-in-Time) compiler profiles a running program to identify its hotspots. Hotspots are then translated into native code, for efficient execution. Using similar approach, we propose a methodology to identify hotspots and exploit their parallelization potential on multicore systems. Proposed methodology selects and parallelizes each DOALL loop that is either contained in a hotspot method or calls a hotspot method. The methodology could be integrated in front-end of a JIT compiler to parallelize sequential code, just before native translation. However, compilation to native code is out of scope of this work. As a case study, we analyze eighteen JGF (Java Grande Forum) benchmarks to determine parallelization potential of hotspots. Eight benchmarks demonstrate a speedup of up to 7.6x on an 8-core system.

Key Words: Just-in-Time Compilation, Loop Level Parallelization, Multicore System, Runtime Analysis, Java Virtual Machine.

1. INTRODUCTION

Multiple cores are typically exploited by parallelizing computer applications in a variety of ways. In case of writing a new application, a convenient approach is to design a parallel algorithm explicitly [1-3]. However, algorithmic re-structuring for existing sequential applications is an on trivial manual effort. Automated parallelization techniques often rely on parallelizing compilers and runtime information. For example, auto-parallelizing compiler Parafrase-2 [4] detects and exploits implicit parallelism using a symbolic analysis framework [5]. Auto-parallelizing compilers typically use heuristics [6] and profiler feedback to analyze and parallelize code by re-compiling [7]. A drawback of static auto-parallelizing compilers is that dynamic execution state of application is not available during compilation. On the other hand, dynamic compilers and run time systems could exploit characteristics of running code in parallelization process.

Runtime systems parallelize applications either speculatively [8-11] or non-speculatively [12-15]. In speculative parallelization, potential parallel tasks are assumed to have no dependences and run using either TLS (Thread Level Speculation) [16] or transactional memory [17]. Results are not committed if the system detects dependence violation(s). Runtime system ensures

* Department of Computer Science & Engineering, University of Engineering & Technology, Lahore.
** Al-Khwarizmi Institute of Computer Science, University of Engineering & Technology, Lahore.

the resolution of dependences by squashing and re-running some of parallel tasks. This is a best effort approach that exploit parallelism if possible, otherwise code is run sequentially. In non-speculative parallelization paradigms, dependences are analyzed first and code is usually transformed to expose hidden parallelism. Parallel tasks are synchronized properly to preserve sequential semantic, and avoid dead/live locks and data races. However, both cases have their own challenges.

JIT systems are typically used to facilitate dynamic compilation of binary code during execution [19-21]. In case of Java, inefficiency of interpreted Java code stimulated the renaissance of JIT technologies [19]. Java (source code) compiler converts source code into bytecode which is stored in class file format. Classes are loaded in JVM (Java Virtual Machine) on-demand and bytecode instructions are interpreted by JVM. For JIT compilation, JVM profiles running applications to select most frequently called and/or most time consuming code regions as hotspots. JIT compiler dynamically compiles hotspots to potentially optimized native code. Since JIT compilers can exploit runtime characteristics of applications, it is plausible to use JIT compilation infrastructure for parallelization.

Typically, majority of computer applications spend large amount of their runtime in the hotspots [22-23]. We observed that compute intensive hotspots have huge parallelization potential [22]. This work focus on a single goal: achieve whatever parallelism can be realized from sequential code without any effort on the part of exploring hidden parallelism. Being a best effort approach, it may improve scalability where it can exploit parallelism potential but in other cases it may not modify even a single loop. Using profiler feedback, compute intensive DOALL loops are selected from Java bytecode just as JIT compiler selects frequently executing code for native translation. We have two reasons for considering loop level parallelization in this context. First, we observed that by setting a threshold on application's execution time, we are left with only a few most time consuming methods [22]. For example, setting 90% threshold in JGF Crypt benchmark revealed that a single method consumed 90% time of the application [24]. Such cases are not suitable for method level parallelization even on dual core system. Similarly, JIT compilation infrastructure selects only few methods as hotspots. Method level parallelization determines potential parallelism by doing inter-procedural analysis of complete application. During inter-procedural analysis, if some non-hotspot method is found as a caller of hotspot(s), modifications will also be needed in the non-hotspot method. Eventually, we will be dealing with entire application and taking almost no advantage of JIT compilation infrastructure. In contrast, modifications applied at loop level remains local to the hotspot only. JIT compiler could produce parallel native code transparently.

The paper is organized in following sections: Section 2 presents related work. Problem statement is formulated in Section 3 along with qualitative and quantitative features. Overall methodology is proposed in Section 4. Parallelization steps and implementation details are given in Section 5. Case studies and results are discussed in Section 6. Paper is concluded in Section 7.

2. RELATED WORK

Bytecode level parallelization has been tried since the inception of Java language [18]. However, due to lack of instrumentation and on-the-fly class modification APIs, the effort relied on static modifications of single class at a time without considering profiler feedback. Now-a-days, JIT parallelization is being revisited, thanks to the proliferation of multicore/manycore systems and advancements in virtualization technologies [25-28,30]. Österlund and Löwe exploit JVM's garbage collector to support JIT parallelization [26-28]. A merger of DBP (Dynamic Binary Parallelization) and TLS is presented to emphasize the limitations of DBP and difficulties involved in JIT parallelization [29]. Leung et. al. proposed auto-parallelizing extensions for Java JIT compiler so that the compiler could find potentially parallelizable code and compile it for parallel execution on multicore CPU and GPGPU (General Purpose Graphic Processing Unit) [30]. However, code generation depends on RapidMind and GPU hardware [31]. Majority of other efforts on runtime parallelization focus on speculative execution and/or exploit method level parallelism [32-38].

3. PROBLEM FORMULATION

Let an application calls N_m methods during execution and each method m_j consists of k loops, where $j \geq 1$ and $k \geq 0$. Starting from main() method, j-1 other methods are typically called in hierarchical manner and inter-procedural relationships are represented as a call graph. Call graph is a directed graph G = <V, E>, where V is a finite set of vertices and E is a finite set of edges. Each vertex $v \in V$ represents a method invocation and each edge $e \in E$ between a vertex pair (u,v) represents one or more invocations of v by u (i.e. $u \rightarrow v$). Static call graph is constructed by source code browsing whereas dynamic call graph is obtained by profiling the running application. Sorted flat profile F is a list representation of dynamic call graph, where $|F| = N_m$. Typically, F also contains runtime information like calls count, time consumption and percentage time consumption of each method. Percentage time consumption of a method is actually PC (Percentage Contribution) of method toward total execution time of application, where PC is defined as:

$$PC = \frac{\text{Net Time Consumed by the Method}}{\text{Total Time Consumed by the Appliction}} \times 100$$

3.1 Percentage Contribution Threshold

T_{PC} (Percentage contribution threshold) is the part of application run time ($\leq$ 100%) that we want to be parallelized [22]. For example, setting T_{PC} = 80% for an application means that we are interested in parallelizing only most time consuming methods (i.e. hotspots) that collectively consume 80% time of the application. Fig. 1 shows the effect of setting T_{PC} = 90% for eighteen JGF application benchmarks [24], where N_h is the number of hotspots. It is obvious from Fig. 1 that majority of methods are shunt out because they collectively consume $\leq$10% time of the application. Analyzing and modifying these methods is likely to increase runtime overhead and may result in performance degradation compared to sequential code.

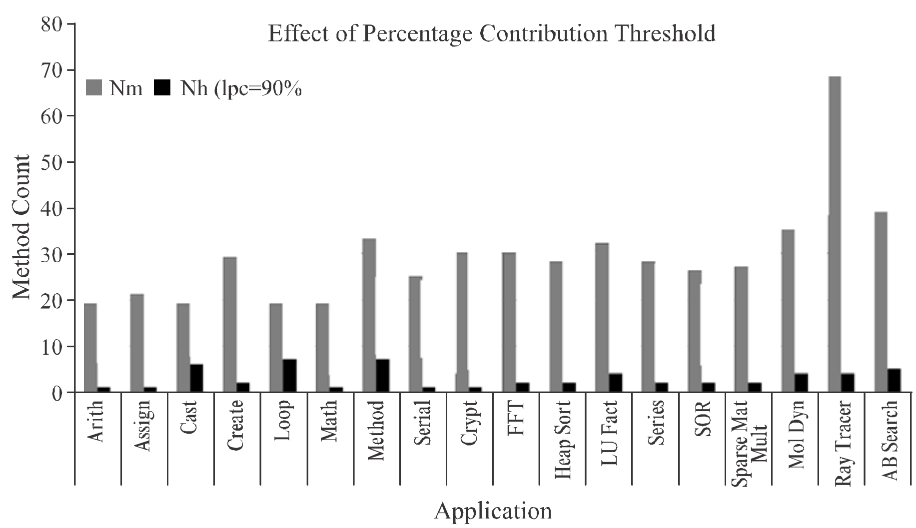


FIG. 1. SELECTION OF HOTSPOTS USING T_{PC} = 90%

T_{PC} facilitates the selection of hotspot methods. Next, we need to determine various characteristics of hotspot methods. We enumerate these characteristics in catalogs of qualitative and quantitative features of methods, as shown in Tables 1-2, respectively.

3.2 Qualitative Features of Methods

Qualitative features are binary variables to represent different characteristics of the method. Each qualitative feature indicates the presence (or absence) of a specific characteristic of a method, as described in Table 1. For example, LOOPY=0 means that the method does not contain loops. The idea of qualitative features is inspired by Nano-patterns that were proposed to characterize and classify Java methods [39]. Catalog of qualitative features is constructed by extended catalog of Nano-patterns from 17 to 32, and giving them compact and descriptive names. Previously, we used qualitative features to analyze thread level speculative parallelization potential at runtime [22]. We showed that binary features are very important decisive factors for runtime qualitative analysis of parallelization potential of methods. Qualitative features are generic in nature and could be used in any software reverse engineering and reengineering activity. We used some relevant features in this work.

3.3 Quantitative Features of Methods

Presence of a particular characteristic of method potentially necessitates the quantification of that characteristic. For example, if a method contains loops (i.e. LOOPY=1), we need to determine the number of single and nested loops. For this, we will observe the quantitative features f_{37} and f_{38} in Table 2. In Table 2, 15 quantitative features are cataloged to represent static and dynamic characteristics of a method. Static and dynamic characteristics are gathered by parsing classes at load time and profiling the running application, respectively. Qualitative and quantitative features abstract the general purpose code characteristics to help in runtime code comprehension. In this work, we used only those features that are helpful in loop level parallelization. Each feature is determined by using a specific algorithm. For the sake of brevity, only two algorithms, related to determination of f_{37} and f_{38}, are presented in section 4.2.

TABLE 1. QUALITATIVE FEATURES OF METHODS

ID	Feature	If True then the Method…
f_0	NO_ARGS	Takes no arguments
f_1	VALUE_ONLY_ARG-S	Takes only pass-by-value arguments
f_2	REF_ONLY_ARGS	Takes only pass-by-reference arguments
f_3	MIXED_ARGS	Takes mixed any arguments
f_4	ARRAY_ARGS	Takes one or more array arguments
f_5	NO_RET	Returns void
f_6	VALUE_RET	Returns primitive value
f_7	REF_RET	Returns reference value
f_8	STATIC	is static
f_9	RECUR	is recursive
f_{10}	LOOPY	contains at least one loop
f_{11}	NESTED_LOOPY	contains at least one nested loops
f_{12}	EXCEPT	throws exception
f_{13}	LEAF	Has no callee method
f_{14}	OBJ_C	creates new objects
f_{15}	FIELD_R	reads class field(s)
f_{16}	FIELD_W	writes class field(s)
f_{17}	TYPE_M	uses type casting
f_{18}	NO_BR	has straight line code
f_{19}	LOCAL_R	reads local variable(s)
f_{20}	LOCAL_W	writes local variable(s)
f_{21}	ARRAY_C	creates new array(s)
f_{22}	MDARRAY_C	creates new multi-D array(s)
f_{23}	ARRAY_R	reads array value(s)
f_{24}	ARRAY_W	writes array value(s)
f_{25}	THIS_R	reads field value(s) of 'this' object
f_{26}	THIS_W	writes field value(s) of 'this' object
f_{27}	OTHER_R	reads field value(s) of other object(s)
f_{28}	OTHER_W	writes field value(s) of other object(s)
f_{29}	SFIELD_R	reads static field value(s)
f_{30}	SFIELD_W	writes static field value(s)
f_{31}	SAMENAME	calls overloaded method(s)

4. PROPOSED METHODOLOGY

Proposed methodology transforms Java classes at load time and works in three phases. Overall work flow is shown in Fig. 2. In profiling phase, an application is test-run to get profiling data. Profiler output is fed back to JVM during actual run. Using a value of T_{PC} (i.e. 90% in this paper), top N_h hotspot methods are selected form the flat profile F which is sorted by PC in descending order. JVM class loader is hooked so that classes could be parsed and transformed at load time [22]. Each class is parsed and modified just before it is loaded by the JVM. In parsing phase, list of methods L_m of a class i is acquired to determine if it contains a hotspot. If a method m_{ij} is hotspot, it is parsed to generate (1) list of qualitative features (2) list of quantitative features (3) list of backward jumps L_{SL} (4) IR tuples, and (5) instruction patterns. A list of nested loop L_{NL} is then generated using the loops of L_{SL}. In modification phase, a heuristic on call count (CC i.e. feature f_{46}) of m_{ij} is used to determine the potential location of parallelizable loop(s). If CC<N_m and m_{ij} is LOOPY then potentially parallelizable loop(s) lies within m_{ij} otherwise lies within some caller of m_{ij}. This heuristic implies that if CC is significantly large, the time consumption of m_{ij} is not due to the loops in it but (potentially) it has been called within a loop of its parent method. In later case, parent of m_{ij} becomes a hotspot provided that it is LOOPY. In any case, we get a loop l_{ijk}. If l_{ijk} is DOALL, it is marked to be modified for parallel execution using the operations mentioned in modification phase of Fig. 2 and threading framework of section 4.4.

TABLE 2. QUANTITATIVE FEATURES OF METHODS

ID	Feature	Description
f_{32}	FIELDs	#Non-static fields accessed in method body
f_{33}	SFIELDs	#Static fields accessed in method body
f_{34}	CALLs	#Method calls in the method
f_{35}	JUMPS	#Jumps (jump instructions) in the method
f_{36}	BRANCHES	#Forward jumps (branches) in the method
f_{37}	SINGLELOOPS	#Single loops in the method
f_{38}	NESTEDLOOPS	#Nested loops in the method
f_{39}	ICOUNT	#Instructions in the method
f_{40}	LOOPICOUNT	#Instructions in the loop bodies
f_{41}	STACKMAX	Maximum stack slots (i.e. stack size)
f_{42}	LOCALMAX	#Local variables (including arguments)
f_{43}	ARGS	#Arguments of the method
f_{44}	TIME	Time consumed by the method
f_{45}	PC	Percentage Contribution of the method
f_{46}	CC	Call Count of the ethod

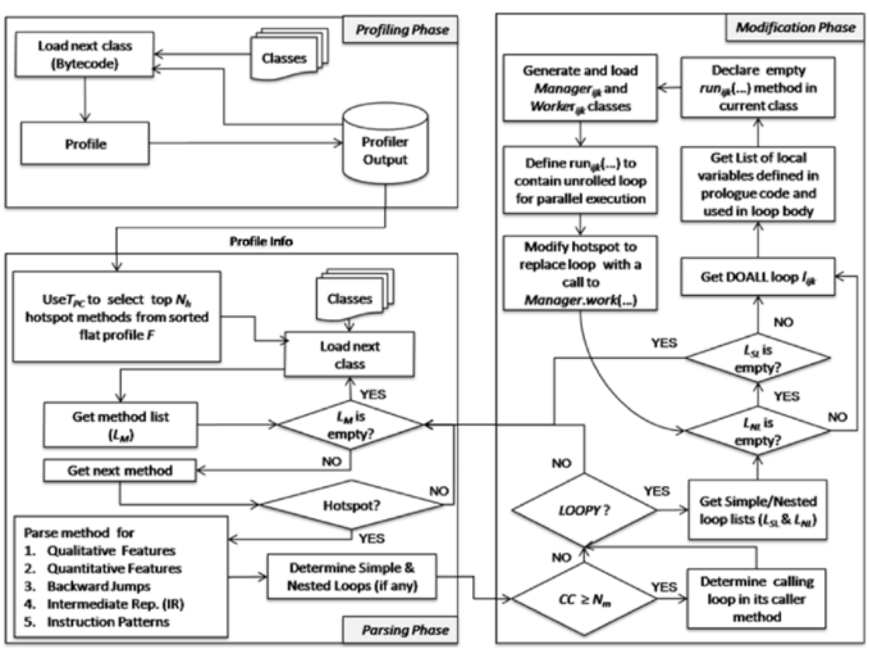


FIG. 2. WORKFLOW OF PROPOSED PARALLELIZATION METHODOLOGY

4.1 Parallelization Criteria

There are two criteria for best effort parallelization of a loop.

Criterion-1: Hotspot Selection: Set T_{PC} = 90% and select most time consuming methods that collectively consume 90% time of application, as hotspots.

Criterion-2: Loop Selection: If a hotspot has significantly high CC value (e.g. $\geq N_m$), then go to its calling method(s). In (any of) calling method, if the hotspot is called in a loop and the loop is DOALL, transform it for parallel execution.

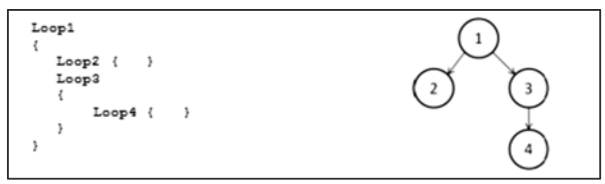
(i) Otherwise, if the hotspot itself contains DOALL loop(s), transform it (them) for parallel execution.

(ii) In case of invalidation of (I) and (II), run unmodified sequential application.

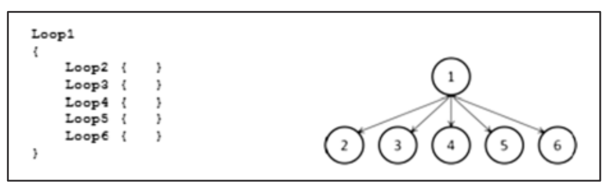
4.2 Loop Profiling

Loop profiling is used to determine the features like SIMPLELOOPS and NESTEDLOOPS. In each hotspot, loops are detected by recording the backward jumps. Each backward jump is represented as quadruple <Offset, Target, Index, Stride>, where Offset is the offset of backward jump, Target is offset of the target label of backward jump, Index is the variable acting as loop index and Stride is the step size of loop iterations. All backward jumps are recorded during parsing phase. Each backward jump is a potential single loop. A backward jump is one whose target has already been visited [39], either in terms of labels or memory addresses. Labels are used in bytecode because exact memory addresses are not known in intermediate code. By constructing basic block level CGF (Control Flow Graph), we can classify a backward jump as a loop if its Target lies in one of the dominator blocks of the block that contains its Offset. A block d dominates a block b (i.e. d DOM b), if all paths from entry block to bincluded. Also, DOM (b) denotes a set of all nodes that dominate b (including b itself).

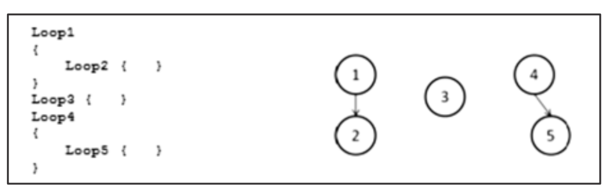
Nested loops are determined by observing the organization of simple loops. If a loop lies exactly within another loop then we come up with a loop nest. For two simple loops l_i and l_j if $Offset_i > Offset_j$ AND $Target_i < Target_j$ then l_j lies within l_i. So, there exist a 2-level nested loop instead of two single loops. In real world code, inner loops in a loop nest may appear in a variety of ways, as shown in Fig. 3. A loop nest could be represented as a loop tree to accommodate all possible organizations of inner loops. Root of tree represents the outer most loop and other nodes represent inner loops of root. The data associated with each node is the loop quadruple, a reference to its parent node and a list of references to its children nodes. Traversing nodes of a loop tree, we can represent nested loops as a 5-tuple <Offset, Target, Nest-Level, {Index-Vector}, {Stride-Vector}> where Offset is the offset of outer most loop, Target is offset of target label of outer most loop, Nest-Level is the height of loop tree, Index-



(a) TRANSFORM_INTERNAL() METHOD OF JGF FFT BENCHMARK



(b) RUNITERS() METHOD OF JGF MOLDYN BENCHMARK. A LOOP FOREST IS IN



(c) MATGEN() METHOD OF LUFACT BENCHMARK

FIG.. 3. LOOP TREES IN

Vector is a list of indices of all loops in loop nest and Stride-Vector is a list of step sizes of all loop in loop nest. Generally, a loop forest containing single and/or multi-node tree(s), is constructed against each hotspot.

4.2.1 Algorithm for Identification of Single Loops

Single loop detection algorithm is shown in Table 3. Let S_i be the instruction stream of a method. During interpretation in a test run, each visited label l is added to a list of visited labels L_v. For each branch instruction b, if the branch's target label l_b has already been visited then b represents a backward jump. Let $Block_A$ and $Block_B$ are two basic blocks (as nodes) in CFG. If $b \in block_A$ and $l_b \in block_B$ and $block_B \in$ DOM ($block_A$), then b is a loop conditional. Prepare quadruple <Offset, Target, Index, Stride> against b and add to a list of single loops L_{loop}.

4.2.2 Algorithm for Loop Forest Construction

Once we get a list of single loops L_{loop}- using the algorithm shown in Figure 4, we can determine nested loops by using algorithm shown in Table 4. Considering each single loop $l_s \in L_{loop}$ as a node, loop tree T_l is constructed against each nested loop and added to a loop forest F_l. Depending upon the availability of loops, F_l could possibly be (1) empty (2) containing single-node tree(s) only (3) containing multi-node tree(s), or (4) containing a mixture of single-node and multi-node trees. At start the loop forest F_l is empty and a tree T_l is constructed using the first loop of L_{loop} as root node. Subsequent loops from L_{loop} are either added to an existing tree or cause the generation of new tree(s). An existing tree is re-adjusted if an outer loop comes after some inner loop(s) so that outer most loop is always the root node.

TABLE 3. ALGORITHM TO IDENTIFY SINGLE LOOPS

```
Input: S_i
Output: L_loop
FOR EACH instruction i
IF i == l THEN
        Add l to L_v
END IF
IF i == b AND l_b ∈ L_v THEN
          Backward Jump found.
          IF b∈ block_A AND
l_b∈ block_B AND
block_B ∈ DOM (block_A)
THEN
                    Prepare Quadruple <Offset, Target, index, stride>
                    Add <Offset, Target, index, stride> to L_loop
END IF
END IF
END FOR
```

4.3 Loop Classification

Using feature f_{37} and f_{38}, we can iterate on all loops to classify them. As we are only interested in parallelization of compute intensive DOALL loops (having arbitrary stride size), we select DOALL loops by observing potential inter-iteration data dependences. Data dependences are analyzed by recognizing instruction patterns corresponding to read/write of local variables, arrays elements, and class members of primitive and user-defined data types. In a DOALL loop, all memory access (instruction) patterns operate on independent memory locations in each iteration. As number of instruction patterns depends on instruction set size, we define an intermediate representation to reduce the (instruction) pattern processing cost.

TABLE 4. ALGORITHM TO CONSTRUCT LOOP FOREST. MULTI-NODE TREES IN THE FOREST REPRESENT NESTED LOOPS

```
Input: L_loop
Output: F_l
FOR EACH l_s∈ L_loop
IF F_l is empty THEN
          Create a new tree rooted at l_s, in F_l
ELSE Identify an existing T_l in F_l
IF T_l found THEN
IF l_s is inner loop of root of T_l THEN
             Insert l_s to T_l at appropriate place
ELSE reorder T_l to make l_s its root
             END IF-ELSE
ELSE create a new tree rooted at l_s, in F_l
         END IF-ELSE
END IF-ELSE
END FOR
```

4.3.1 Intermediate Representation of Bytecode Instructions

IR (Intermediate Representation) of bytecode instructions is defined to reduce instruction pattern count and potential pattern processing effort. If an instruction set contains n instructions. We might have to look for $(n)^p$ combinations to recognize an instruction pattern of length p. These combinations could be reduced if we reduce n by symbolically representing n instructions with m symbols, where m<n. For example, a subset of bytecode instructions {IADD, LADD, FADD, DADD} is used to perform arithmetic addition of two {integer, long-integer, floating–point, double-precision-floating-point} numbers, respectively. A high level IR symbol ADD could suffice to recognize any of these four instructions. Similarly, we can recognize entire instruction set using a smaller set of IR symbols. By defining IR symbols, we could represent ~200 bytecode instructions (i.e. $n \approx 200$) with 42 symbols (i.e. $m = 42$), as shown in Table 5. Labels are typically induced by compiler to facilitate control flow and demarcation of basic blocks. We consider LBL as part of IR symbols because labels are integral part of compiled code. As elaborated in next sub-section, presentation of instruction patterns in terms of IR symbols increases the occurrence frequency of instruction patterns. Using IR symbols, we have about five times (i.e. $\lceil n/m \rceil$) fewer choices at each position in instruction pattern.

4.3.2 Recognition of Instructions Patterns

Compilers typically generate an instruction pattern against each source code statement. Java source compiler generates a stream of bytecode instructions which is interpreted by JVM. We recognize bytecode instruction patterns to distinguish memory accesses. The idea starts with the preparation of a catalog of ISA-specific fundamental instruction patterns. Each fundamental pattern consists of at least two instructions in a specific order and performs a smallest indivisible source level task e.g. "variable initialization". Some instructions like INC or LV (Table 5) could independently perform an indivisible source level task e.g. "j++;". We enumerate such instructions as independent instructions. A pattern is an arrangement of two or more independent instructions. Figure 6 shows an inner loop from SORrun(…) method of JGF SOR benchmark [24], to elaborate instruction pattern recognition.

Source code and bytecode of the loop is shown in Fig. 4(a-b), respectively. Fig. 4(b) also shows the IR tuple <Symbol, Opcode, [Argument(s)]> against each instruction, where Symbol is IR symbol (defined in Table 5), Opcode is the opcode of encountered instruction and optional Argument(s) represents zero or more arguments of the instruction. IR tuple of a label does not contain any opcode and its Argument contains string representation of actual label. For compact representation, IR symbols of an instruction pattern are concatenated, as shown in Table 4. For example, read operation on Gim1[j] in Fig. 4(a) was translated into bytecode instructions at line 10, 11, 12 of Fig. 4(b). Using IR symbols, we can represent this instruction pattern as LR-LV-LVA. In Fig. 4 (b), all occurrences of LR-LV-LVA pattern are encircled with dashed lines. LR-LV-LVA is a fundamental pattern because it is composed of instructions only and indivisible into sub-patterns. All fundamental patterns and partial pattern components are recognized and assigned unique IDs P_{xy} and C_{xy}, respectively, as shown in Table 6, where each P_{xy} (or C_{xy}) represents a pattern (or pattern component) y having x level composition. Composition level of fundamental patterns is zero. Using IDs of fundamental patterns and pattern components, and independent instructions, parse tree of bytecode, shown in Fig. 4(b), is shown in Fig. 5. It is constructed in reverse direction taking leaves at level 0. First level composite patterns do not contain any other composite pattern. Second level composition contains at least one first level composite pattern, third level contains at least one 2nd level composite pattern, and so on. Each leaf is either an ID of fundamental pattern or pattern component, or an independent instruction, as shown in Fig. 5.Each non-leaf node represents a composite pattern and its composition depends on the level blow it. A composite pattern may consist of independent instructions,

TABLE 5. INTERMEDIATE REPRESENTATION OF BYTECODE INSTRUCTIONS

Symbol	Description	Bytecode Instructions
_	Do Nothing	NOP
LC	Load Constant	ACONST_NULL, ICONST_M1, ICONST_0, ICONST_1, ICONST_2, ICONST_3, ICONST_4, ICONST_5, LCONST_0, LCONST_1, FCONST_0, FCONST_1, FCONST_2, DCONST_0, DCONST_1, BIPUSH, SIPUSH, LDC, LDC_W, LDC2_W
LV	Load Value	ILOAD, LLOAD, FLOAD, DLOAD
LR	Load Reference	ALOAD
LVA	Load Value from Array	IALOAD, LALOAD, FALOAD, DALOAD, BALOAD, CALOAD, SALOAD
LRA	Load Reference Array Value	AALOAD
SV	Store Value	ISTORE, LSTORE, FSTORE, DSTORE
SR	Store Reference	ASTORE
SVA	Store primitive Array Value	IASTORE, LASTORE, FASTORE, DASTORE, BASTORE, CASTORE, SASTORE
SRA	Store Reference Array Value	AASTORE
PP	Pop	POP, POP2
DP	Duplicate	DUP, DUP_X1, DUP_X2, DUP2, DUP2_X1, DUP2_X2
SP	SWAP	SWAP
AO	Arithmetic Operation	IADD, LADD, FADD, DADD, ISUB, LSUB, FSUB, DSUB, IMUL, LMUL, FMUL, DMUL, IDIV, LDIV, FDIV, DDIV, IREM, LREM, FREM, DREM
LO	Logical Operation	INEG, LNEG, FNEG, DNEG, ISHL, LSHL, ISHR, LSHR, IUSHR, LUSHR, IAND, LAND, IOR, LOR, IXOR, LXOR
INC	Increment	IINC
P2P	Primitive-Primitive Casting	I2L, I2F, I2D, L2I, L2F, L2D, F2I, F2L, F2D, D2I, D2L, D2F, I2B, I2C, I2S
CMP	Compare	LCMP, FCMPL, FCMPG, DCMPL, DCMPG
IF1	1-Value IF Statement	IFEQ, IFNE, IFLT, IFGE, IFGT, IFLE
IF2	2-Values IF Statement	IF_ICMPEQ, IF_ICMPNE, IF_ICMPLT, IF_ICMPGE, IF_ICMPGT, IF_ICMPLE, IF_ACMPEQ, IF_ACMPNE
GJR	Unconditional Jump	GOTO, JSR, RET
SW	Switch Statement	TABLESWITCH, LOOKUPSWITCH
RV	Return Value	IRETURN, LRETURN, FRETURN, DRETURN
RR	Return Reference	ARETURN
VD	Void	RETURN
LSF	Load Static Field	GETSTATIC
SSF	Store Static Field	PUTSTATIC
LF	Load Class Field	GETFIELD
SF	Store Class Field	PUTFIELD
INV	Invoke a Method	INVOKEVIRTUAL, INVOKESPECIAL, INVOKESTATIC, INVOKEINTERFACE, INVOKEDYNAMIC
NW	Create New Object	NEW
NVA	Create New Value Array	NEWARRAY
NRA	Create New Array of Objects	ANEWARRAY
@	Array Length	ARRAYLENGTH
XCP	Throw Exception	ATHROW
CCH	Check Cast	CHECKCAST
IOF	Instance of	INSTANCEOF
ME	Monitor Enter	MONITORENTER
MX	Monitor Exit	MONITOREXIT
NMA	Create New n-D Array	MULTIANEWARRAY
IFN	If Statement (Compares Null)	IFNULL, IFNONNULL
LBL	Label Induced by Compiler	

fundamental patterns and its children composite patterns (Table 6). The root of the tree represents top level composite pattern that is entire bytecode region shown in Fig. 4(b).

```
for (int j=1; j<Nm1; j++) {
    Gi[j] = omega_over_four *
      (Gim1[j] + Gip1[j] + Gi[j-1] + Gi[j+1])
                 + one_minus_omega * Gi[j];
}
```

FIG. 4(a). SOURCE CODE OF A LOOP TAKEN FROM SORRUN(...) METHOD OF JGF SOR BENCHMARK

	Bytecode	IR tuple
1	L16	<LBL, L16>
2	ICONST_1	<LC, 4>
3	ISTORE 17	<SV, 54, 17>
4	L17	<LBL, L17>
5	GOTO L18	<GJR, 167, L18>
6	L19	<LBL, L19>
7	ALOAD 14	<LR, 25, 14>
8	ILOAD 17	<LV, 21, 17>
9	DLOAD 6	<LV, 24, 6>
10	ALOAD 15	<LR, 25, 15>
11	ILOAD 17	<LV, 21, 17>
12	DALOAD	<LVA, 49>
13	ALOAD 16	<LR, 25, 16>
14	ILOAD 17	<LV, 21, 17>
15	DALOAD	<LVA, 49>
16	DADD	<AO, 99>
17	ALOAD 14	<LR, 25, 14>
18	ILOAD 17	<LV, 21, 17>
19	ICONST_1	<LC, 4>
20	ISUB	<AO, 100>
21	DALOAD	<LVA, 49>
22	DADD	<AO, 99>
23	L20	<LBL, L20>
24	ALOAD 14	<LR, 25, 14>
25	ILOAD 17	<LV, 21, 17>
26	ICONST_1	<LC, 4, 1>
27	IADD	<AO, 96>
28	DALOAD	<LVA, 49>
29	DADD	<AO, 99>
30	DMUL	<AO, 107>
31	DLOAD 8	<LV, 24, 8>
32	ALOAD 14	<LR, 25, 14>
33	ILOAD 17	<LV, 21, 17>
34	DALOAD	<LVA, 49>
35	DMUL	<AO, 107>
36	DADD	<AO, 99>
37	L21	<LBL, L21>
38	DASTORE	<SVA, 82>
39	L22	<LBL, L22>
40	IINC 17	<INC, 132, 17>
41	L18	<LBL, L18>
42	ILOAD 17	<LV, 21, 17>
43	ILOAD 11	<LV, 21, 11>
44	IF_ICMPLT L19	<IF2, 162, L19>

FIG. 4(b.) BYTECODE AND ITS IR TUPLES

4.3.3 Inter-Iteration Data Dependence

DOALL loops could be identified by making sure that loop iterations either does not contain any instruction patterns corresponding to memory access or they access independent memory locations. We need to identify instruction patterns that are used to read/write local variables, arrays elements and class members (i.e. fields) of both primitive and user-defined types. If a loop does not contain any instruction

TABLE 6. INSTRUCTION PATTERNS IN EXAMPLE LOOP.

ID	Instruction Pattern	Composition
P_{00}	LBL-LC-SV	Fundamental Patterns
P_{01}	LBL-GJR	
P_{02}	LR-LV-LVA	
P_{03}	LR-LV-LC-AO-LVA	
P_{04}	LBL-LR-LV-LC-AO-LVA	
P_{05}	LBL-INC	
P_{06}	LBL-LV-LV-IF2	
C_{00}	LBL-LR-LV	Pattern Components
C_{01}	LBL-SVA	
P_{10}	LR-LV-LVA-LR-LV-LVA-AO	P02-P02-AO
P_{11}	LV-LR-LV-LVA-AO	LV-P02-AO
P_{20}	LR-LV-LVA-LR-LV-LVA-AO-L-R-LV-LC-AO-LVA-AO	P10-P03-AO
P_{30}	LR-LV-LVA-LR-LV-LVA-AO-L-R-LV-LC-AO-LVA-AO-LBL-L-R-LV-LC-AO-LVA-AO	P20-P04-AO
P_{40}	LV-LR-LV-LVA-LR-LV-LVA-A-O-LR-LV-LC-AO-LVA-AO-L20--LR-LV-LC-AO-LVA-AO-AO	LV-P30-AO
P_{50}	LV-LR-LV-LVA-LR-LV-LVA-A-O-LR-LV-LC-AO-LVA-AO-LB-L-LR-LV-LC-AO-LVA-AO-AO--LV-LR-LV-LVA-AO-AO	P40-P11-AO
P_{60}	LBL-LR-LV-LV-LR-LV-LVA-LR--LV-LVA-AO-LR-LV-LC-AO-LV-A-AO-LBL-LR-LV-LC-AO-LV-A-AO-AO-LV-LR-LV-LVA-AO--AO-LBL-SVA	C00-P50-C01
P_{70}	LBL-LC-SV-LBL-GJR-LBL-LR--LV-LV-LR-LV-LVA-LR-LV-LVA--AO-LR-LV-LC-AO-LVA-AO-L-BL-LR-LV-LC-AO-LVA-AO-A-O-LV-LR-LV-LVA-AO-AO-LB-L-SVA-LBL-INC-LBL-LV-LV-I-F2	P00-P01-P60-P05-P-06

pattern corresponding to inter-iteration data dependences, it is DOALL loop because of independent iterations. Let's analyze the loop given in Fig. 4 to determine if it is DOALL or not. Source code and Bytecode of the loop (Fig. 4) reveals that the only variables involved are local because compiled code does not contain any bytecode instruction related to class members (Fig. 4(b)). Table 7 shows the types and compiler-assigned indices of variables used by bytecode instructions. For example, loop index j is indexed at 17 and could be determined from IINC instruction. In Table 6, we can see that only one write operation, represented by P_{60}, is performed in each iteration. This pattern has sixth level composition and its first component C_{00} contains information about the variable involved. The IR tuples of C_{00} are <LBL, L19>, <LR, 25, 14>, <LV, 21, 17> at line 6-8. It shows that the variable is indexed at 14 which is "double[] Gi". Hence, we are concerned about the read/write patterns of array elements. Write operation of Gi depends on three read operations of Gi, one of which is performed in the same iteration and is harmless. Other two reads in an iteration j are performed in immediately previous iteration j-1 and next iteration j+1, which causes inter-iteration data dependences. The patterns of reading Gi[j-1] and Gi[j+1] are P_{03} at line 17-21 and P_{04} at line 23-28, respectively. Hence, the loop in Figure 6 is not DOALL so could not be parallelized without resolving dependences.

4.4 Threading Framework

A threading mechanism is required by JIT compiler to modify selected loops for parallelization execution. We designed a Java threading framework to be generated directly in bytecode according to the characteristics of

TABLE 7. VARIABLES USED IN EXAMPLE LOOP

Variable	Name	Type	Index
Local	J	int	17
	Nm1	int	11
	Gi	double[]	14
	omega_over_four	double	6
	Gim1	double[]	15
	Gip1	double[]	16
	one_minus_omega	Double	8

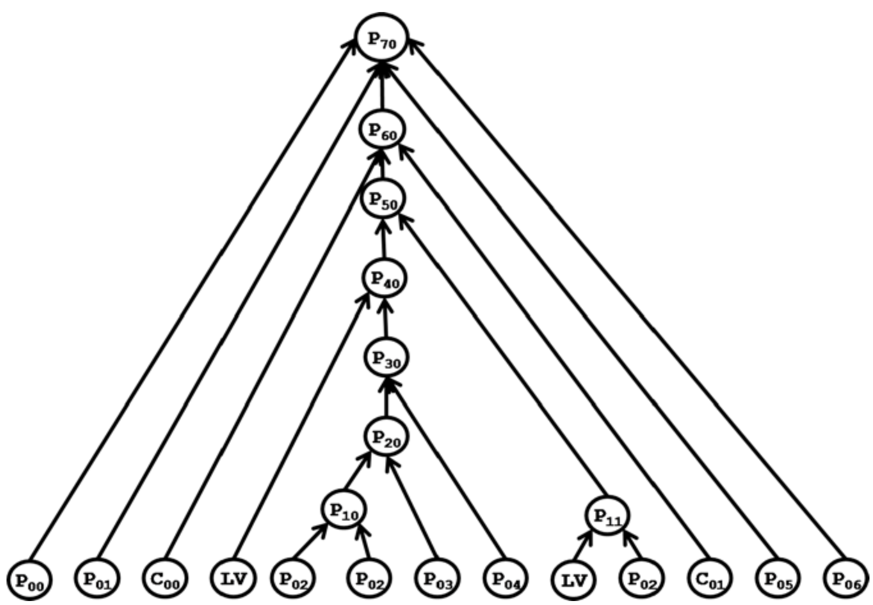


FIG. 5. PARSE TREE OF EXAMPLE LOOP'S BYTECODE IN TERMS OF INSTRUCTION PATTERN IDS, PATTERN COMPONENT IDS AND INDEPENDENT INSTRUCTIONS

workload. We adapted the idea of source code level JAVAR framework [40]. Our framework consists of only two classes, $Worker_{ijk}$ and $Manager_{ijk}$, that are dynamically generated for each candidate loop l_{ijk}. We used ASM [41] for generation of framework classes (in bytecode) as dynamic part of classes would not be available at compile time [41]. $Worker_{ijk}$ encapsulates the entire implementation of parallel task whereas $Manager_{ijk}$ is responsible for creation and orchestration of workers. $Manager_{ijk}$ contains only one static method work(…) and each candidate loop l_{ijk} is replaced with just a single call to $Manager_{ijk}$.work(…). Fig. 6 shows the interaction of threading framework with $Class_i$ that contain loop l_{ijk} in its method m_{ij}. For a loop l_{ijk}, a single $Manager_{ijk}$ manages life cycle of n $Worker_{ijk}$ threads. Each $Worker_{ijk}$ calls run_{ijk}() method that is defined in $Class_i$. The loop l_{ijk} is replaced with a call to $Manager_{ijk}$. work(...). $Class_i$ makes jxk calls for k DOALL loops in j methods of this class. Fig. 6 shows a cyclic dependency that could be removed by declaring run_{ijk}() before generating $Worker_{ijk}$ and providing its definition after the generation of $Manager_{ijk}$. Actual usage of framework is elaborated in Section 5 using the code in Fig. 8.

4.5 Motivational Example

To demonstrate the step-by-step working of proposed methodology, we identify and parallelize the most suitable loop of JGF Series benchmark [24]. This benchmark manipulates various transcendental and trigonometric functions to calculate Fourier coefficients of function f(x) = $(x+1)^x$. About 10,000 coefficients are computed with an interval of 0.2. Methodology starts with profiling phase in which we found that the application calls 28 methods i.e. N_m = 28. By setting T_{PC} = 90%, we found 2 potential hotspots. For a potential hotspot, top-ranking value of PC is either due to its high CC (Call Count) or due to having compute intensive loops indicated by f_{10}, f_{11}, f_{37}, f_{38} features. The reason is that PC is based on the self-time consumed by a method i.e. time consumption of its callee methods is excluded. Looking at Table 8, we come to know that CC value of both methods is significantly high but only TrapezoidIntegrate() method contains one single loop. Hence, high time consumption (i.e. 99.9% collectively) of these methods is due to high call count and not due to the loops in their own code. To determine the immediate caller methods, we have to look at the

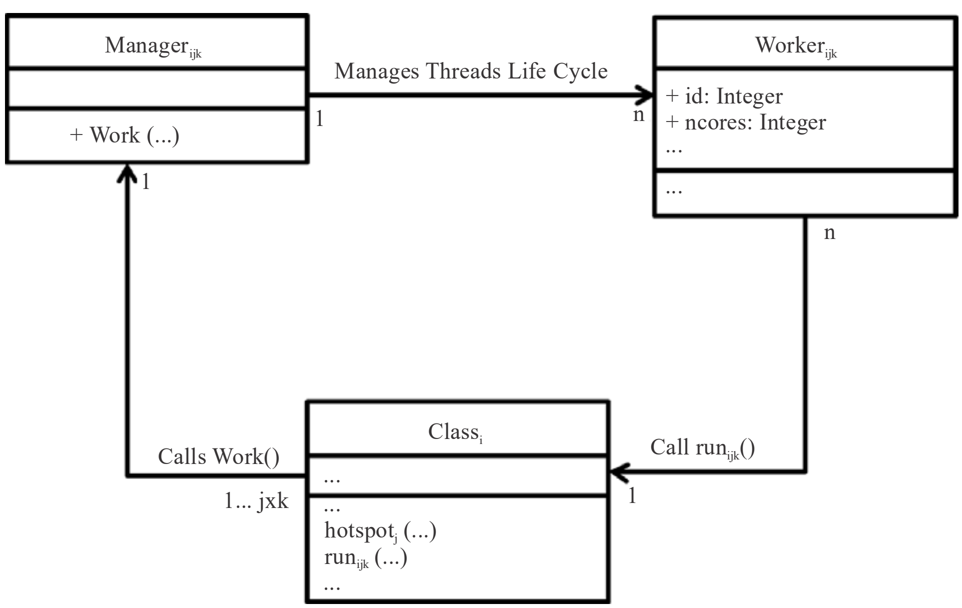


FIG. 6. CLASS DIAGRAM SHOWING THE ASSOCIATION OF THREADING FRAMEWORK CLASSES WITH THE CLASS CONTAINING HOTSPOT METHOD

relevant portion of application call graph shown in Fig. 7. It shows that Do()method calls TrapezoidIntegrate() and TrapezoidIntegrate() calls thefunction(). In Table 9, qualitative features show that Do() method is (1) non-static (2) contains single loop(s) only (3) calls other methods (not leaf in call graph) (4) does not create any object and 1-D or n-D array (5) reads/writes array elements and local variables (6) only reads non-static class fields, and (7) does not read/write static class fields. Quantitative features say that Do() methodis (1) called only once (2) contains one single loop (3) reads two non-static class fields (4) has five call sites, and (5) reads/writes up to four local variables. Although self-time consumption of Do() method is 0.1%, it calls two most time consuming methods in a single loop and itself is called once. Hence, the loop in it is exploitable for parallelize execution.

TABLE 8. FEATURES OF POTENTIAL HOTSPOTS IN JGF SERIES

Type	Name	Trapezoid Integrate	The Function
Qualitative	STATIC	0	0
	LOOPY	1	0
	NESTED_LOOPY	0	0
	LEAF	0	0
	PC	60.70%	39.20%
	CC	19999	19999000
	CALLs	3	5
	SINGLELOOPS	1	0
	NESTEDLOOPS	0	0
	LOCALMAX	15	6

```
...
|===|=== JGFkernel()V
|===|===|=== Do()V
|===|===|===|=== startTimer(Ljava/lang/String;)V
|===|===|===|===|=== start()V
|===|===|===|=== TrapezoidIntegrate(DDIDI)D
|===|===|===|===|=== thefunction(DDI)D
|===|===|===|===|===|===Math.pow(DD)
|===|===|===|=== stopTimer(Ljava/lang/String;)V
|===|===|===|===|=== stop()V
|===|=== JGFvalidate()V
...
```

FIG. 7. RELEVANT PORTION OF CALL GRAPH OF SERIES BENCHMARK. IT SHOWS THAT TRAPEZOIDINTEGRATE() CALLS THEFUNCTION() AND ITSELF CALLED BY DO(). INTER-PROCEDURAL RELATIONSHIPS ARE PRESENTED USING BAR-TAB "|===" E.G. JGFKERNEL() IS IMMEDIATE PARENT OF DO() BUT SIBLING OF JGFVALIDATE()

5. IMPLEMENTATION DETAILS

Implementation details include the steps taken to parallelize a candidate loop and a short note on proof of concept. All modifications are done on bytecode, as elaborated in section 4.

5.1 Parallelization Steps

Modifications steps are explained here in terms of Java source code. Bytecode level implementations details are given in section 5.2.

Loop Extraction: The loop is shown at line 7-10 of Fig. 8(a) in source code of Do() method. Bytecode of this loop is extracted from the method and represented as IR tuples to recognize instruction patterns for data dependence analysis.

Data Dependence Analysis: Bytecode of Do() method contains instruction patterns of local variable read/write. Besides loop index i, one local variable omega is defined

TABLE 9. RELEVANT FEATURES OF DO() METHOD OF JGF SERIES

Qualitative Features		Quantitative Features	
Feature	Value	Feature	Value
STATIC	0	PC	0.10%
LOOPY	1	CC	1
NESTED_LOOPY	0	FIELDs	2
LEAF	0	SFIELDs	0
OBJ_C	0	CALLs	5
FIELD_R	1	SINGLELOOPS	1
FIELD_W	0	NESTEDLOOPS	0
LOCAL_R	1	LOCALMAX	4
LOCAL_W	1		
ARRAY_C	0		
MDARRAY_C	0		
ARRAY_R	1		
ARRAY_W	1		
SFIELD_R	0		
SFIELD_W	0		

before the loop body and used in loop body. Local variables omega and i are not written in the loop body so there is no inter-iteration data dependence due to local variables. Table 9 shows that no static field is read/written and non-static fields are read but not written. However, arrays are read/written but source code does not show any array read. The bytecode reveals that in TestArray[][] write, TestArray[] is first loaded on stack and then its TestArray[][i] element is written. There is no data dependence due to TestArray[][i] because it is independently written in each iteration and without involving a read. Hence, the loop is DOALL and we can parallelize it.

Declaration of Run$_{ijk}$() Method: A method run$_{ijk}$ is declared in the class of Do() method, as shown in Fig. 8(b), where a, b, c are <start, end, step> tuple for a worker thread. We cannot define run$_{ijk}$ yet because <start, end, step> is calculated in dynamically generated partitionLoop() method of Worker$_{ijk}$ class. We just declare run$_{ijk}$ here so that a call in Worker$_{ijk}$ could not pop error.

Generation of Worker$_{ijk}$ and Manager$_{ijk}$ Classes: Next step is to generate and load Worker$_{ijk}$ and Manager$_{ijk}$ classes. We observed that all classes have to be loaded by the same class loader as that of the application. Against the source code shown in Fig. 8(c-d), bytecode is generated using ASM [41].

Definition of Run$_{ijk}$() Method: Due to cyclic dependency shown in Fig. 6, we define run$_{ijk}$() after code generation for Worker$_{ijk}$ and Manager$_{ijk}$ classes. Fig. 8(b) shows this definition, where calculation of a, b, c depends on the number of workers created in Manger$_{ijk}$ i.e. kept equal to number of CPU cores as shown in Fig. 8(d).

Loop Replacement in Hotspot: Finally, the loop in Do() method is replaced with a single call to Manager$_{ijk}$.work() method as shown on line 7 of Fig. 8(e). Original loop and its replacement is encircled by dotted line to highlight in Fig. 8 (a) and Fig. 8(e), respectively.

5.2 Proof of Concept

As a proof of concept, we implemented a research prototype by extending SeekBin [22]. As a Java agent, it hooks JVM's class loader, captures classes loading into JVM, and manipulates bytecode just before loading. SeekBin reads sorted flat profile F to determine the classes to be manipulated. Classes are parsed, transformed and generated (i.e. Manager$_{ijk}$, Worker$_{ijk}$) using ASM bytecode engineering library and loaded using java.lang.instrument API. The tool can profile and parse any sequential application to generate qualitative and quantitative features, IR tuples, instruction patterns, loop profiling, class generation and loading etc.

6. CASE STUDIES

Data is collected by profiling and parsing eighteen benchmark applications [24] to analyze their parallelization potential. Data is analyzed for code comprehension regarding exploitable parallelism.

6.1 Code Comprehension

The purpose of code comprehension is twofold: first, we want to explore the parallelization potential of the application at hand. To avoid additional runtime overhead, it is crucial to estimate the feasibility of applying proposed methodology. We also need to decide the locality and extent of transformations needed as we want to transform bare minimum amount of most promising code. Table 10 represents an estimate of parallelization potential of 18 benchmarks in terms of method level features. Parallelization potential of an application depends on the number of methods called during execution (N_m), frequency of method calls, number of loops, number of instructions in loop bodies, and dependencies among loop iterations. However, not all methods and loops are potentially feasible for parallelization and we need to filter them out by setting suitable T_{PC} value i.e. T_{PC} = 90% in this case. As a result, we converge to only few methods as potential hotspots.

```
void Do() {
        double omega;
        JGFInstrumentor.startTimer("Section2:Series:Kernel");
        TestArray[0][0] = TrapezoidIntegrate((double)0.0, (double)2.0, 1000,(double)0.0,0)/(double)2.0;
        omega = (double) 3.1415926535897932;

        for (inti = 1; i<array_rows; i++) {
                TestArray[0][i] = TrapezoidIntegrate((double)0.0,(double)2.0,1000,omega * (double)i,1);
                TestArray[1][i] = TrapezoidIntegrate((double)0.0,(double)2.0,1000,omega * (double)i,2);
        }

        JGFInstrumentor.stopTimer("Section2:Series:Kernel");
}
```

(a) SOURCE CODE OF DO() METHOD OF SERIES BENCHMARK

```
void run_ijk(int a, int b, int c, double omega){
for (inti = a; i< b; i = i+c) {
        TestArray[0][i] = TrapezoidIntegrate((double)0.0,(double)2.0,1000,omega * (double)i,1);
        TestArray[1][i] = TrapezoidIntegrate((double)0.0,(double)2.0,1000,omega * (double)i,2);
  }
}
```

(b) DEFINITION OF RUN$_{ijk}$

```
public class Worker_ijk implements Runnable{
int ID, ncores, a, b, c, fr, to, step;
SeriesTesttc;
      double l1;
Worker_ijk(SeriesTest cls, int aa,
int bb, int cc, intnc, int id, double v1){
tc = cls;   ID = id;        ncores = nc;
      a = aa;     b = bb;     c = cc;
      l1 = v1;
      }
      private void partitionLoop(){
                              step = c;
                              intblk = (b + ncores-1)/ncores;
                              fr = ID*blk;
                              if(ID == 0) fr = ID*blk+1;
                              to = (ID+1)*blk;
                              if (to > b ) to = b;
      }    public void run() {
                partitionLoop();
tc.run_ijk(fr,to,step, l1);
      }
}
```

(c) DEFINITION OF WORKER$_{ijk}$ CLASS

```
public class Worker_ijk implements Runnable{
int ID, ncores, a, b, c, fr, to, step;
SeriesTesttc;
      double l1;
Worker_ijk(SeriesTest cls, int aa,
int bb, int cc, intnc, int id, double v1){
tc = cls;   ID = id;        ncores = nc;
      a = aa;     b = bb;     c = cc;
      l1 = v1;
      }
      private void partitionLoop(){
                              step = c;
                              intblk = (b + ncores-1)/ncores;
                              fr = ID*blk;
                              if(ID == 0) fr = ID*blk+1;
                              to = (ID+1)*blk;
                              if (to > b ) to = b;
      }    public void run() {
                partitionLoop();
tc.run_ijk(fr,to,step, l1);
      }
}
```

(d) DEFINITION OF MANAGER$_{ijk}$ CLASS

```
void Do() {
        double omega;
        JGFInstrumentor.startTimer("Section2:Series:Kernel");
        TestArray[0][0]=TrapezoidIntegrate((double)0.0, (double)2.0, 1000,(double)0.0,0)/(double)2.0;
        omega = (double) 3.1415926535897932;

        Manager_ijk.work(this, 1, array_rows, 1, omega);

        JGFInstrumentor.stopTimer("Section2:Series:Kernel");
}
```

(e) LOOP REPLACEMENT IN DO() METHOD

FIG. 8. STEPS OF JUST-IN-TIME PARALLELIZATION

6.2 Parallelization of JGF Benchmarks

Thirteen benchmark applications are explicitly transformed and eight benchmarks showed a reasonable speedup, as shown in Fig. 9. Instead of exposing hidden parallelism in other benchmarks, proposed best effort approach prefers to restore sequential versions of applications that do not show speedup. To demonstrate the scalability of transformed applications, we passed "number of workers" as command line argument, instead of getting it from target system as mentioned in Fig. 8(d). Transformed applications are run on an 8-core system comprising 2 x Quad Core Intel® Xeon® E5405, 1333 MHz FSB, CPU Speed 2.0 GHz, L1 D Cache 32 KB, L1 I Cache 32 KB, L2 Cache 2x(2x6) = 24 MB and 8 GB DRAM. In order to assess the scalability, data is organized in two sets; long running and short running applications, as shown in Fig. 9(a-b).

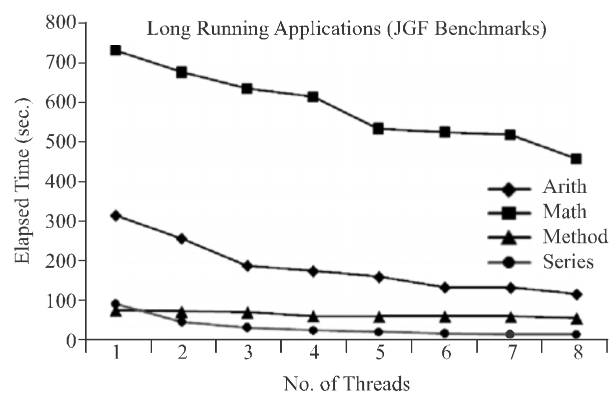
6.2.1 Long Running JGF Benchmarks

Four long running benchmarks that demonstrated speedup are Series, Arith, Math and Method, as shown in Fig. 9(a). Parallelization of JGF Series benchmark has been described in section 4.5. Series benchmark demonstrated a speedup of 6.9x, which is comparable to HP (Hand Parallelized) version of Series, shown in Fig. 10(d). Outer loops contain instructions related to getting system time. They cannot be multithreaded without generating additional code for thread-local time management (which is out of scope of this work). Speedup observed in Arith, Math and Method benchmarks is 2.7x, 1.6x and 1.4x, respectively. Although speedup of Math and Method is not quite significant on an 8-core system, the point is that changes are not permanent. In case of unsatisfactory speedup, we can restore to sequential execution anytime because transformations are applied at runtime and code on disk is intact.

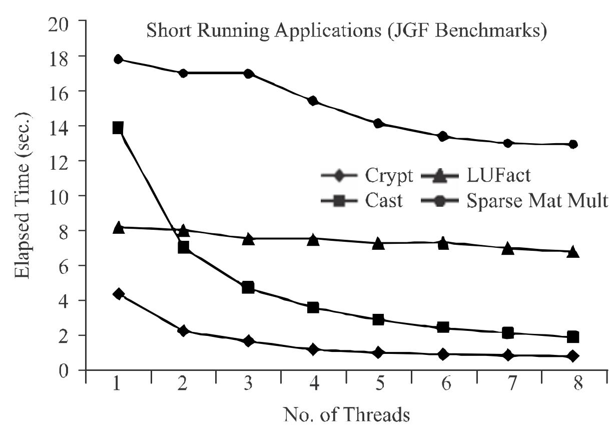
6.2.2 Short Running JGF Benchmarks

Short running benchmarks that showed speedup are Crypt, LUFact, SparseMatMult and Cast, as shown in Fig. 9(b). In Crypt, out of 30 methods, only one method cipher_idea() consumes 90% time when called twice in the application, as shown in Table 10. In cipher_idea(), there is no single loop and one 2-level nested loop. Nested loop is DOALL and its outer loop is parallelized.

Crypt demonstrated a speedup of 5.8x and perfectly scale with the increasing number of threads, as shown in Fig. 9(b). HP version of JGF Crypt, when run on the same system, demonstrated 7x speedup and resembling scalability, as shown in Fig. 10(c). The result is quite encouraging because proposed methodology is



(a) LONG RUNNING

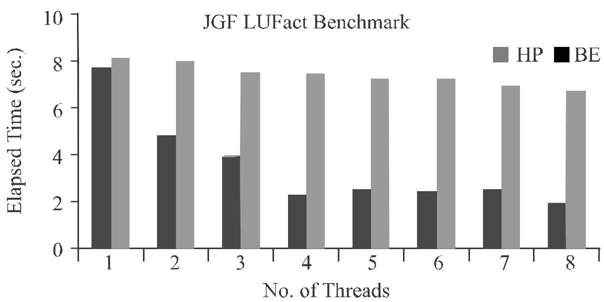


(b) SHORT RUNNING BENCHMARKS

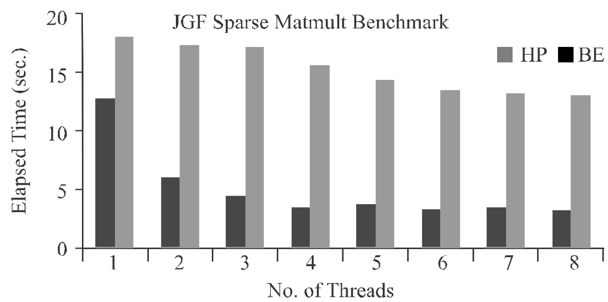
FIG. 9. RESULTS OF APPLYING BEST EFFORT JIT PARALLELIZATION. SCALABILITY OF JGF

transforming code on-the-fly. In LUFact, only 4 out of 32 methods consume 90.8% time. LUFact contains 15 single and 4 nested loops. However, selected 4 methods contain 4 single and 1 nested loops (collectively), as shown in Table 10. Most time consuming method dgefa() is called once and contains one 2-level nested loop. Method daxpy() and idamax() are called in inner and outer loops of dgefa()'s nested loop, respectively. Outer loop is parallelized to achieve a speedup of 1.2x on 8-core system. On the same system, the speedup is not encouraging as compared to 4x speedup of HP JGF LUFact, as shown in Fig. 10(a). Looking at the code of HP version, we observed that this version achieved speedup by using barrier construct at four locations to synchronize the threads. This type of flexibility is not supported yet in our approach.

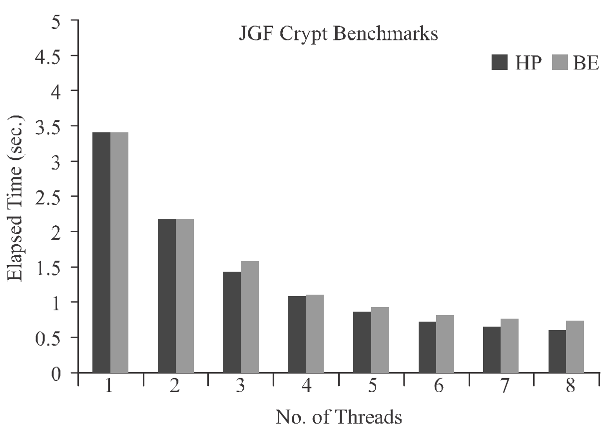
SparseMatMult calls 27 unique methods but only two methods consumed 90.1% time in a single call each, as shown in Table 10. Most time consuming method test() contains one single and one 2-level nested loop and second method JGFinitialise() contains one single loop. There is no harmful data dependences in all 3 loops, however, single loops contain trivial amount of computation. On parallelizing all 3 loops, we observed performance degradation as compared to sequential version. By parallelizing only nested loop of test(), we observed the scalability shown in Fig. 9(b), with a speedup of 1.4x. Running HP version on the same system, we observed a speedup of 4.1x. Scalability comparison of both versions is given in Fig. 10(b). HP version achieves this speed up by restructuring the implemented algorithm. For proper load balancing, signature of hotspot test() is



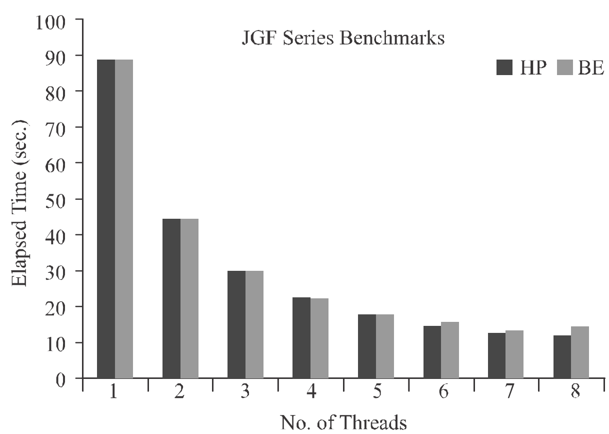
(a) LUFACT BENCHMARK



(b) SPARSEMATMULT BENCHMARK



(c) CRYPT BENCHMARK



(d) SERIES BENCHMARK. HP MIGHT OCCASIONALLY INVOLVE ALGORITHM RESTRUCTURING

FIG. 10. COMPARISON OF BEST EFFORT (BE) RESULTS WITH THAT OF HAND PARALLELIZED VERSIONS OF JGF

changed to control the nested loop partitioning from outside the hotspot. Due to the reasons mentioned in section 1, proposed methodology works locally (i.e. within a hotspot only) without altering the interface (i.e. signature) of hotspot methods

Cast benchmark called 19 methods and by setting T_{PC} = 90%, we converged to 6 methods that collectively consume 91.7% time of application (Table 10). Starting from most time consuming method JGFrun(), we found 4 nested loops here and this method is called once. Only a single loop is found in one of other 5 methods i.e., in printperf(). In nested loops, compute intensive code was found in inner loops that were parallelized. Outer loops contain timing routines and cannot be parallelized due to the reason mentioned in section 6.2.1. JGF Cast demonstrated highest speedup of 7.6x. Overall, the observed speedup is in the range 1.2 - 7.6x.

7. CONCLUSIONS

This work emphasizes that best effort JIT compiler inspired parallelization has great potential of parallelizing executable code at runtime. Loops in compute-intensive applications exhibit greater parallelization potential, which makes it a worthwhile option. Although it may not be able to parallelize each and every application, it is plausible to exploit parallelism without programmer intervention. Best effort exploits parallelism wherever possible and there is no harm because transformations are not made permanent. In case of failure, sequential execution could be restored. However, in case of success, transformations could be made permanent at any time. The main contributions of this paper include: (1) catalogs of qualitative and

TABLE 10. PARALLELIZATION POTENTIAL OF JGF BENCHMARK APPLICATIONS

Benchmark	TPC = 100%						TPC = 90%					
	Nm	f_{46}	f_{37}	f_{38}	f_{39}	f_{40}	Nh	f_{46}	f_{37}	f_{38}	f_{39}	f_{40}
Arith	19	1805	1	12	2843	1929	1	1	0	12	2249	1913
Assign	21	1597	1	10	2802	1916	1	1	0	10	2114	1900
Cast	19	641	1	4	1458	776	6	146	1	4	1033	776
Create	29	1E+08	1	15	3119	2155	2	2E+07	0	15	2455	2139
Loop	19	482	1	3	819	186	7	161	1	3	427	186
Math	19	3875	1	30	5308	3904	1	1	0	30	4714	3888
Method	33	9E+07	1	8	1749	927	7	6E+07	0	8	1106	911
Serial	25	2E+06	9	4	1738	708	1	1	8	4	1142	692
Crypt	30	48	8	1	1966	798	1	2	0	1	390	374
FFT	30	37	5	2	1663	563	2	3	1	1	472	399
HeapSort	28	2E+06	4	1	1079	230	2	1E+06	2	1	156	131
LUFact	32	3E+05	15	4	2169	820	4	3E+05	4	1	482	284
Series	28	2E+07	2	1	1144	158	2	2E+07	1	0	115	22
SOR	26	26	0	3	1058	147	2	2	0	3	218	147
SparseMatMult	27	27	3	1	1080	124	2	2	2	1	187	107
MolDyn	35	4E+05	12	3	3196	1349	4	3E+05	1	1	878	591
RayTracer	68	4E+08	4	2	3092	525	4	4E+08	1	0	260	61
ABSearch	39	7E+07	18	2	3241	782	5	5E+07	4	2	816	354

qualitative features for runtime code comprehension; (2) compact intermediate representation of ISA and instruction pattern recognition for dependence analysis; (3) threading framework; and (4) a set of algorithms to profile and parallelize DOALL loops.

With increasing number of cores per chip, it is now possible to use at least part of this compute power to analyze the runtime characteristics of an application with minimal impact on expected performance. Such information can be exploited to improve the application performance. Such approaches are particularly beneficial for complex long-running applications, which may not be simple to analyze manually. Loops are one of the simplest constructs that can be extracted from any type of code. Our work is an effort to demonstrate the feasibility of this approach. In past efforts, success criteria of an automated or semi-automated parallelization approach has been based on achievable speedup. When compared with manually parallelized applications, these approaches do not fare well because one parallelization technique may work for a few parts of the code but degrades others. Restricting to hotspots and ability to reverse parallelization transforms at runtime enhances the possibilities of parallelizing long running compute-intensive applications. By relaxing the speedup requirements, it is possible to try multiple techniques for different parts of application code at runtime to achieve optimal performance with no user input.

8. FUTURE WORK

This work proposes a best effort parallelization methodology that could be used within the front end of JIT (i.e. dynamic) compiler. Integration of this methodology in an actual dynamic compiler is the obvious next step. We have designed a development project to integrate this methodology in an open source JIT compiler.

ACKNOWLEDGEMENT

This research was conducted during Ph.D. study of first author, at University of Engineering & Technology, Lahore, Pakistan.

REFERENCES

[1] Zhang, T.Y., and Suen, C.Y., "A Fast Parallel Algorithm for Thinning Digital Patterns", Communications of the ACM, Volume 27, No. 3, pp. 236-239, 1984.

[2] Alfredo Buttaria, J.L., Kurzaka, J., and Dongarra, J., "A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures", Parallel Computing Volume 35, No. 1, pp. 38-53, 2009.

[3] Wang, Y., Fan, J., Liu, W., and Han, Y., "A Parallel Algorithm to Construct BISTs on Parity Cubes", IEE Proceedings of 2nd International Conference on Information Science and Control Engineering, pp. 54-58, 2015.

[4] Polychronopoulos, C.D., Girkar, M., Haghighat, M.R., Lee, C.L., Leung, B., and Schouten, D., "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors", International Journal of High Speed Computing, Volume 1, No. 1, pp. 45-72, 1989.

[5] Haghighat, M., and Polychronopoulos, C., "Symbolic Program Analysis and Optimization for Parallelizing Compilers", Springer Berlin Heidelberg, pp. 538-562, 1993.

[6] Whaley, J., and Kozyrakis, C., "Heuristics for Profile-Driven Method-Level Speculative Parallelization", Proceedings of International Conference on Parallel Processing, pp. 147-156, 2005.

[7] Tournavitis, G., Wang, Z., Franke, B., and O'Boyle, M.F.P., "Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping", ACM SIGPLAN Notices, Volume 44, No. 6, pp. 177-187, 2009.

[8] Jang, H., Kim, C., and Lee, J.W., "Practical Speculative Parallelization of Variable-Length Decompression Algorithms", ACM SIGPLAN Notices, Volume 48, pp. 55-64, 2013.

[9] Jimborean, A., Clauss, P., Martinez, J.M., and Sukumaran-Rajam, A., "Online Dynamic Dependence Analysis for Speculative Polyhedral Parallelization", Euro-Par, Parallel Processing, pp. 191-202, 2013.

[10] Liu, B., Zhao, Y., Li, Y., Sun, Y., and Feng, B., "A Thread Partitioning Approach for Speculative Multithreading", The Journal of Supercomputing, Volume 67, No. 3, pp. 778-805, 2014.

[11] Yiapanis, P., Rosas-Ham, D., Brown, G., and Lujan, M., "Optimizing Software Runtime Systems for Speculative Parallelization", ACM Transactions on Architecture and Code Optimization, Volume 9 No. 4, 2013.

[12] Alle, M., Morvan, A., and Derrien, S., "Runtime Dependency Analysis for Loop Pipelining in High-Level Synthesis", Proceedings of 50th Conference on Annual Design Automation, pp. 51:1-51:10, ACM, 2013.

[13] Aumage, O., Barthou, D., Haine, C., and Meunier, T., "Detecting Simdization Opportunities through Static/ Dynamic Dependence Analysis", Euro-Par: Parallel Processing Workshops, Lecture Notes in Computer Science, Volume 8374, pp. 637-646, 2014.

[14] Tzenakis, G., Papatriantafyllou, A., Vandierendonck, H., Pratikakis, P., and Nikolopoulos, D., "BDDT: Block-Level Dynamic Dependence Analysis for Task-Based Parallelism", Advanced Parallel Processing Technologies, Lecture Notes in Computer Science, Volume 8299, pp. 17-31, 2013.

[15] Verdoolaege, S., Carlos Juega, J., and Cohen, A., "Polyhedral Parallel Code Generation for CUDA", ACM Transactions on Architure Code Optimum, Volume 9 No. 4, pp. 54:1-54:23, 2013.

[16] Steffan, J.G., Colohan, C.B., Zhai, A., and Mowry, T.C., "A Scalable Approach to Thread-Level Speculation", ACM, Volume 28, No. 2, pp. 1-12, 2000.

[17] Harris, T., Cristal, A., Unsal, O.S., Ayguade, E., Gagliardi, F., Smith, B., and Valero, M., "Transactional Memory: An Overview", IEEE Micro, Volume 27, No. 3, pp. 8-29, 2007.

[18] Bik, A.J., and Gannon, D., "A Prototype Bytecode Parallelization Tool", Concurrency - Practice and Experience, Volume 10, Nos.11-13, pp. 879-885, 1998.

[19] Aycock, J., "A Brief History of Just-in-Time", ACM Computing Surveys, Volume 35, No. 2, pp. 97-113, 2003.

[20] Pemberton, S., and Daniels, M., "Pascal Implementation: The P4 Compiler and Interpreter", Ellis Horwood, 1982.

[21] Ierusalimschy, R., DeFigueiredo, L.H., and Celes, W., "The Implementation of Lua 5.0", Journal of Universal Computer Science, Volume 11, No. 7, pp. 1159-1176, 2005.

[22] Mustafa, G., Waheed, A., and Mahmood, W., "SeekBin: An Automated Tool for Analyzing Thread Level Speculative Parallelization Potential", Proceedings of 7th IEEE International Conference on Emerging Technologies, pp. 1-6, 2011.

[23] Knuth, D.E., "An Empirical Study of FORTRAN Programs", Software: Practice and Experience, Volume 1, No. 2, pp. 105-133, 1971.

[24] Mathew, J. A., Coddington, P. D., and Ha-wick, K. A., "Analysis and development of Java Grande benchmarks," Proc. 1999 conference on Java Grande, USA, ACM, pp. 72-80, 1999.

[25] Hinsen, K., "A Glimpse of the Future of Scientific Programming", Computing in Science & Engineering, Volume 15, No. 1, pp. 84-88, 2013.

[26] Österlund, E., "Garbage Collection Supporting Automatic JIT Parallelization in JVM", LNU, 2012.

[27] Österlund, E., and Löwe, W., "Analysis of Pure Methods Using Garbage Collection", Proceedings of ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, pp. 48-57, Beijing, China, 2012.

[28] Österlund, E., "Automatic Memory Management System for Automatic Parallelization", LNU, 2011.

[29] Koch, T.J.K.E.V., and Björn, F., "Limits of Region-Based Dynamic Binary Parallelization", SIGPLAN Notices, Volume 48, No. 7, pp. 13-22, 2013.

[30] Leung, A., Lhotak, O., and Lashari, G., "Automatic Parallelization for Graphics Processing units", Proceedings of 7th International Conference on Principles and Practice of Programming in Java, Calgary, pp. 91-100, Alberta, Canada, 2009.

[31] Monteyne, M., "Rapidmind Multi-Core Development Platform", RapidMind Inc., Waterloo, Canada, February, 2008.

[32] Hammacher, C., Streit, K., Hack, S., and Zeller, A., "Profiling Java Programs for Parallelism", Proceedings of ICSE Workshop on Multicore Software Engineering, pp. 49-55, 2009.

[33] Christopher, J.F.P., Clark, V., and Allan, K., "LIBSPMT: A Library for Speculative Multithreading", Sable Technical Report, 2007.

[34] Pickett, C.J.F., and Verbrugge, C., "SableSpMT: A Software Framework for Analysing Speculative Multithreading in Java", Proceedings 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 59-66, Portugal, 2005.

[35] Carlstrom, B.D., Chung, J., Chafi, H., McDonald, A., Minh, C.C., Hammond, L., Kozyrakis, C., and Olukotun, K., "Transactional Execution of Java Programs", OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Languages, 2005.

[36] Chen, M.K., and Olukotun, K., "The JRPM System for Dynamically Parallelizing Sequential Java Programs", IEEE Micro, Volume 23, No. 6, pp. 26-35, 2003.

[37] Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., and Chew, L.P., "Optimistic Parallelism Requires Abstractions", Communication ACM, Volume 52, No. 9, pp. 89-97, 2009.

[38] Chan, B., and Abdelrahman, T.S., "Run-Time Support for the Automatic Parallelization of Java Programs", Journal Supercomputer, Volume 28, No. 1, pp. 91-117, 2004.

[39] Singer, J., Brown, G., Luján, M., Pocock, A., and Yiapanis, P., "Fundamental Nano-Patterns to Characterize and Classify Java Methods", Electronic Notes in Theoretical Computer Science, Volume 253, No. 7, pp. 191-204, 2010.

[40] Bik A. J. C., and Gannon D. B., "Automatically exploiting implicit parallelism in Java," Concurrency: Practice and Experience, vol. 9, no. 6, pp. 579–619, 1997.

[41] Bruneton, E., Lenglet, R., and Coupaye, T., "ASM: A Code Manipulation Tool to Implement Adaptable Systems", Technical Report, France Telecom R&D, 2002.