High-Level Data-Access Analysis for Characterisation of (Sub)task-Level Parallelism in Java

R. Stahl, R. Paško, F. Catthoor, R. Lauwereins and D. Verkest
IMEC vzw, Kapeldreef 75, B-3001 Leuven, Belgium
richard.stahl@imec.be

Abstract

In the era of future embedded systems the designer is confronted with multi-processor systems both for performance and energy reasons. Exploiting (sub)task-level parallelism is becoming crucial because the instruction-level parallelism alone is insufficient.

The challenge is to build compiler tools that support the exploration of the task-level parallelism in the programs. To achieve this goal, we have designed an analysis framework to evaluate the potential parallelism from sequential object-oriented programs.

Parallel-performance and data-access analysis are the crucial techniques for estimation of the transformation effects. We have implemented support for platform-independent data-access analysis and profiling of Java programs, which is an extension to our earlier parallel-performance analysis framework. The toolkit comprises automated design-time analysis for performance and data-access characterisation, program instrumentation, program-profiling support and post-processing analysis. We demonstrate the usability of our approach on a number of realistic Java applications.

1. Introduction

Future embedded systems confront the designer with multi-processor architectures, which have performance and energy-consumption constraints. For multi-processor systems in particular, two inseparable challenges exist. First, the parallel tasks have to be identified and extracted. Second, in the optimal case, a very good match should exist between the tasks and the architecture resources. Any significant mismatch in critical parts of the application will result in performance loss, a decrease of the resource utilisation, and reduced energy efficiency of the system.

From the designer's point of view, three general approaches exist to solve those challenges: manual, automated and tool supported. In the first case, the designer manually translates the sequential program into an optimised parallel program with respect to the system constraints. This can lead to the most optimal solution, yet the solution is dedicated to a specific problem. In the second case, the designer uses an automated tool, which leads to the easiest solution for the designer, but the limitations on the current state of the art tools do not allow to deal with realistic applications. Currently, the most realistic approach is the third one that supports the designer's manual effort with a number of analysis and transformation tools. The development of these tools is an important step towards more automated parallelism extraction and optimisation for embedded systems.

We propose a transformation framework for extraction and optimisation of task-level parallelism from sequential Object-Oriented (OO) programs with respect to architectural and energy-consumption constraints. Such sequential OO programs are becoming the most common form of code that is produced for embedded multi-media applications today (in C++ or Java).

As a crucial basis for our framework, we have implemented parallel performance and data-access analysis tools for Java programs. They automatically instrument the code with respect to the designer's input constraints, profile the program, and interpret the profiling information. The tools help the designer to understand the behaviour of a sequential or parallel program, to find the bottlenecks in execution, and to interpret the analysis results.

The main contribution of our parallel performance analysis framework is that it provides the designer with application-oriented platform-independent characterisation of concurrent object-oriented programs, including the effects of data communication. The framework uses the underlying host platform to execute the program, while the parallel program execution and the effects of such an execution are simulated in a virtual parallel machine on top of the host. The machine creates the environment for (seemingly) parallel program execution (Figure 1). The machine represents an abstract model of an arbitrary target platform, defined by a set of plat-

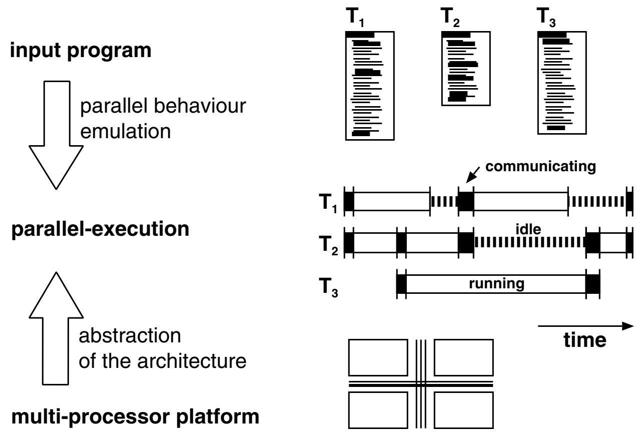

Figure 1. Concept of parallel-execution time. The concept allows simulation of the parallel program execution, communication and idleness of its tasks. Moreover, it abstracts characteristic features of the underlying platform.

form characteristics (task creation, task execution and task communication). Thus, the performance analysis provides high-level platform-independent evaluation of the parallel program performance.

The remainder of this paper is organised as follows. Section 2 gives a concise overview of related work and shows the distinguishing features of our approach. Section 3 describes the parallel performance framework we have implemented; Section 4 describes the data-access analysis part of the framework; Section 6 presents the experimental evaluation of our tools, and finally, Section 7 gives concluding remarks.

2. Related work

The data-access and communication analysis for parallel programs has been already researched in a number of projects. We can identify two main categories: data-access analysis for single-processor platforms with memory hierarchy and communication analysis for parallel programs on multi-processor platforms.

We have been partially inspired by work of Ding and Zhong [4], who introduce platform-dependent, run-time monitoring of data accesses. This approach is based on compiler-directed instrumentation of single-threaded C programs. Moreover, the authors implemented the concept of selective monitoring to improve the performance of the run-time system.

A similar approach for data-access analysis has been introduced by Bormans et al. [5] They use design-time data-access analysis to identify all possible data-accesses in the sequential C programs. Afterwards, the executable specification is profiled and the data-access traces generated.

Leeman et al. [6] introduce a technique for data-access profiling for power estimation. They use method-level data-access summaries, which are inserted into the program code at design-time so that the run-time system can gather the data-access traces for arbitrary data types.

We distinguish from these approaches in the following: first, we have introduced the concept of parallel execution, which allows the designer to perform parallel program analysis without the previous mapping to the target platform, and second, we have introduced the concept of parallel communicating tasks for which we analyse the computation as well as communication cost.

In the area of parallel systems, the research focus has been mainly on communication analysis and optimisation. These approaches usually require explicit communication between the tasks. However, few approaches, targeting compilation for shared or distributed-memory systems, consider the communication analysis as a crucial part of the performance metrics.

Miller et al. [12] have introduced Paradyn - parallel performance measurement tools. It focuses on the profiling and post-processing of profile information for long-running large-scale programs written in high-level data-parallel languages. Paradyn uses a dynamic instrumentation technique based on constraints given by the designer, while providing an interface between the low-level platform-dependent objects and high-level language features. Haake et al. [13] have introduced a similar approach, but they have implemented profiling support for the Split-C programs with Active Messages. It is based on fine grain communication profiling while the program traces are post-processed off-line.

Another approach, implemented by Vetter [11] analyses the performance of parallel programs with message passing communication. The main contribution of this work is in the classification of communication inefficiencies, i.e., it is a post-processing phase of performance analysis that gives the designer concise and interpreted performance measures.

Chakrabarti, et al. [7] introduce communication analysis and optimisation techniques for High-Performance Fortran programs. Even though the approach includes performance analysis, the main focus is on the optimisation of the global program communication.

We distinguish from the previous approaches by introducing automated data-access analysis support for high-level programming languages. Additionally, these approaches are intended for a platform-specific performance analysis for particular machines, as opposite to our platform-independent analysis.

We believe that the approach introduced by Tseng [8] is one of the closest to our work. The technique focuses on communication analysis for machine-independent High-

Performance Fortran programs, and provides application-oriented analysis of the communication in the parallel programs. We, on the other hand, introduce design-time data-access analysis for high-level concurrent object-oriented programs. Moreover, we introduce the above mentioned concept of parallel-execution environment with support for performance and data-access profiling.

3. Parallel-performance analysis framework for Java

The proposed performance analysis tool is based on a concept of parallel-execution time (Figure 1), which allows one to abstract specific architectural features of the platform. The concept is used to simulate parallel execution of program tasks while the program is actually executed on the underlying platform, which does not need to be the final target platform [18]. It allows one to reason about the parallelism and communication effects between the tasks with respect to the physical parallelism of the target platform.

We have defined three platform optimisation criteria - task-creation overhead, balanced task granularity and communication overhead. The task-creation overhead is the ratio between the task-creation interval and task execution time. It defines the minimal task granularity, as demonstrated for our approach in the experiment of subsection 6.1. We have elaborated in detail on the balanced-task execution in our previous work [18].

Herein, we focus mainly on the task communication overhead (Section 4). It represents the amount of time a task spends in transferring data. This time has to be negligible compared to task execution time as it also determines the overall performance of the parallel program. This is demonstrated in the experiments of subsection 6.2.

The output of the performance analysis framework allows one to trade off those features of the parallel program with respect to each other. Thus, the designer can explore trade-offs in communication with respect to task execution time, in task execution time with respect to communication overhead.

We have implemented parallel performance analysis for concurrent Java programs [18]. The tools work as follows. Firstly, the program is automatically transformed based on designer's input constraints. This phase consists of two complementary transformations: performance analysis and data-access analysis, which are combined together to provide complete information on the overall performance of the program. Secondly, the parallel program execution is simulated and profiled. Finally, the profiling information is analysed and interpreted to provide the designer with a more convenient form of profiling output.

We define a data-access model (Figure 2) that consists of main execution thread, number of separate threads and

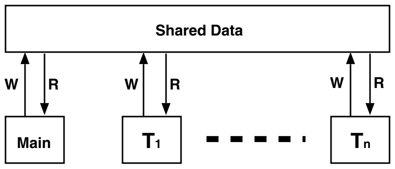

Figure 2. Data-access model: a model of an abstract architecture, used in the design-time data-access analysis to separate non-local thread accesses to shared data from its local ones.

shared data. The design-time data-access analysis resolves all possible shared data which has to be fetched by a thread to enable its correct execution. The implementation of the transformation passes is based on the SOOT optimisation framework [1]. We use the SOOT framework for program analysis (method call graph construction, points-to analysis) that serves as a base for our transformations.

We use the Profiler API as an interface between the design-time instrumentation phase and the run-time profiling support. The profiler gathers information on program execution as well as data-accesses. The profiling support for parallel programs is implemented as an extension to a Java Virtual Machine (JVM) implementation.

The profile information is later parsed by the post-processing phase of the performance analysis. The post-processing consists of a critical-path analysis algorithm, as presented in our previous work [18] and the analysis of data-access traces. This information is essential for further exploitation of task-level parallelism.

4. Design-time data-access analysis and instrumentation

The data-access model (Figure 2) is used as a representation for modelling the accesses to the data shared between different program tasks. Therefore, it serves as the conceptual base for the design-time data-access analysis.

The data-access model consists of the following components: main-program thread, separate threads/tasks and shared data. The main program thread represents the main flow of the program execution. Moreover, all shared data belong to this thread are stored in the shared-data section. The separate threads execute independently of the main thread. They require an amount of data to be read from the shared-data section before they can proceed with execution. On the other hand, the threads generate an amount of data to be stored and later accessed by other parts of the program. All this data is of interest for our platform-independent analysis.

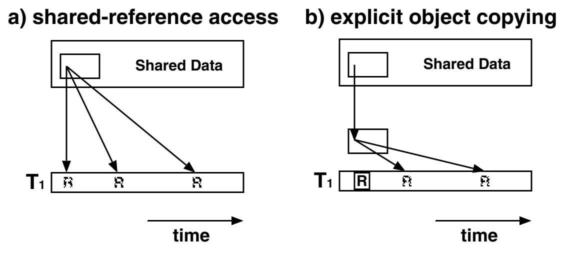

Figure 3. Reference-type data access: a) accesses to shared reference-type data are not taken into account, while b) explicit copying of the object data is counted.

```
doMethodRead(M):
  cloneIfNotTopLevel(M)
  for each local M' invoked in M:
    doMethodRead(M')       // -- LocalCalls
  for each static M' invoked in M:
    doMethodRead(M')       // -- StaticCalls
  for each read parameter P
    resolveTypeRead(M,P)  // -- Params
  for each read member F
    addToMembersList(M,F) // -- Members
    resolveTypeRead(M,F)
```

Figure 4. Algorithm for forward data-read analysis

The designer defines the set of separate threads, and the data-access analysis identifies all possible accesses performed by these threads. To achieve this, the data-access analysis traverses the corresponding parts of the program representation based on a method-call graph. It identifies all the data that are created outside the scope of this thread, and are accessed within its scope. Based on the Java programming language specification [17], access to these non-local data can be performed only via method parameters, class members and return statements. We will refer to these Java program features as thread-shared features. The features created within the scope of a thread are referred to as thread-local features.

We can conceptually split the data-access analysis into two parts: data-read and data-write analysis. The data-read analysis of the thread-shared features works as a pure forward traversal of the program representation, indicating all potential data-read accesses of a particular thread. The data-write analysis traverses the representation forward until it finds any write access to a thread-shared feature. It starts the backward traversal to identify the source of the data assignment and all accesses to this source. In the case of polymorphic method call the analysis resolves and analyses all method call candidates while only one of them is selected and profiled at run-time. Finally, data-access annotations are added for the instrumentation phase. We strictly distinguish the annotation and the instrumentation phase that allows one to implement different instrumentation policies resulting in different types of data-access analysis.

Within the data-access analysis, we generally distinguish three groups of data types: primitive types, reference types, and array types. The primitive-type group consists of char, short, integer, long, float, and double types. The reference-type group holds references to any program classes and arrays. As the Java arrays implement distinct concept within the Java language, we consider them a special (sub-)group with respect to the pure reference type.

The instrumentation phase modifies the program representation with profiler specific code, generating the instrumented program code at the end. The instrumentation uses the policy presented in [6]. Currently, we have implemented support for data-access analysis of array-type data and primitive-type scalars. We identify two possible scenarios for reference-type shared-data accesses (Figure 3): *a)* the accesses to pure reference-type data are not considered as particular data accesses to the shared data region, because the actual accesses only occur at other locations in the code (where they are counted); *b)* explicit copying of the referenced-object members is instrumented though. Once a local copy of shared data is created, the data copying is annotated as shared-data accesses and the later local-copy accesses are not taken into account. This approach can eventually be extended with profiling support for high-level dynamic-data types [6].

The data-access analysis algorithms are implemented recursively because of their recursive nature. Even though we will describe the algorithms separately the final implementation of the tool combines the three phases into a single recursive transformation pass.

4.1. Forward data-read analysis

The data-read analysis identifies the read accesses to the thread-shared features resulting in purely forward recursive traversal of the program representation. The potential candidates for data-read access to thread-shared data are only method parameters and class members (Figure 4).

The analysis consists of the following steps: analysis of local and static method calls, analysis of method parameters and analysis of class members accesses within this method (Figure 4). The algorithm for the analysis starts by entering the representation of one of the designer specified methods (top-level methods):

1. First, a method clone is created for all methods except the top-level methods. This isolates the representation of the sub-graph specified by a top-level method, i.e.,

```
resolveTypeRead(M,P):
  if(P is primitive)      // -- Primitive
    addPrimDataRead(M,P)
  if(P is reference)      // -- Reference
    addRefDataRead(M,P)
    if(P asigned to P')
      resolveRefTypeRead(M,P')
    if(M' invoked on P)
      doMethodRead(M')
  if(P is array)          // -- Array
    if(P assigned to P')
      resolveArrTypeRead(M,P')
    if(element E @idx of P read)
      P is array id
      addArrayDataRead(M,P)
      resolveTypeRead(M,E)
```

Figure 5. Type resolving for data-read analysis

each top-level method is an entrance point to a unique extension of the original method-call graph, avoiding collisions with the others.

2. Second, the algorithm recursively analyses all calls to static and local methods of given class (Figure 4, *LocalCalls, StaticCalls*).

3. Finally, once the flow of the algorithm returns, it analyses accesses to all parameters and later accesses to all class members. As shown in Figure 4 (*Members*), it keeps a database of all members of all classes to avoid feature aliasing, resulting in redundant data-accesses.

The algorithm returns when it reaches the end of *DoMethodRead* method for a top-level method. Eventually, it can enter the same flow for another top-level method. Each top-level method and its sub-graph is identified by a unique indentification tag.

The algorithm enters the *resolveTypeRead* method for each of the method parameters and class members (Figure 5):

- *Primitive type:* all assignments from it are annotated as data-read accesses (*primDataRead*). However, it can also be passed to another method as a parameter. In this case, a particular feature tag is set and the parameter is handled in the called method further. This situation is handled the same way for all three types.
- *Array type:* assignment from array type is annotated as a simple reference-type data-read access. Assignment from its element is annotated as a data-read access to an array structure (*ArrayDataRead*). Each array has a unique identification number, a separate counter at run-time, and an information on array-index accesses.
- *Reference type:* all assignments from reference type or calls to its methods are annotated as potential data-read access (*RefDataRead*). Moreover, the analysis distinguishes the explicit data copying from general shared-data accesses, as already explained before.

```
doMethodWrite(M):
  for each local M' invoked in M:
    doMethodWrite(M')     // -- LocalCalls
  for each static M' invoked in M:
    doMethodWrite(M')     // -- StaticCalls
  for each written parameter P
    resolveTypeWrite(M,P) // -- Params
  for each written member F
    addToMembersList(M,F) // -- Members
    resolveTypeWrite(M,F)
  R = returnStmt(M)       // -- Return
  resolveTypeWrite(M,R)
```

Figure 6. Algorithm for backward data-write analysis

The forward analysis adds annotations to particular code features, while traversing the cloned sub-program representation. These indicators are later used in the instrumentation phase.

4.2. Backward data-write analysis

The backward data-write analysis traverses the representation forward until it finds any write access to a thread-shared feature. Once the feature is identified, it starts the backward traversal to identify the origin of the data assignment. Furthermore, it inspects all potential accesses to this feature. Thus, the data-write analysis uses the forward analysis to resolve all read accesses to thread-local features that can be potentially assigned to thread-shared features. The potential candidates are non-primitive method parameters, class members, and return statements (Figure 6).

The backward analysis consists of the following steps: analysis of local and static method calls, method parameters, class members within this method and return statements. The algorithm for the analysis starts by entering the representation of one of the top-level methods:

1. First, the algorithm recursively analyses all calls to static and local methods of given class (Figure 4, *LocalCalls, StaticCalls*). Those method can assign new values to non-primitive-type shared features (reference-type parameters and members).

2. After returning from the recursive analysis of called methods, the algorithm analyses all write accesses to all parameters and later accesses to all class members. As is shown (Figure 6, *Members*), the algorithm updates the database of all members of all classes to avoid data aliasing.

```
resolveTypeWrite(M,P):
  if(P is primitive)      // -- Primitive
    addPrimDataWrite(M,P)
  if(P is reference)      // -- Reference
    if(P asigned from P')
      top-level-local = findOriginOfRef(M,P')
      if(top-level-local)
        addRefDataWrite(M,P)
    if(M' invoked on P)
      doMethodWrite(M')
  if(P is array)          // -- Array
    if(P assigned from P')
      resolveArrTypeWrite(M,P')
    if(element E @idx of P written)
      P is array id
      addArrayDataWrite(M,P)
      resolveTypeRead(M,E)
      resolveTypeWrite(M,E)

findOriginOfRef(M,R'):    // -- Backward
  if(R' is local)
    return top-level-local
  if(R' is member)
    resolveTypeRead(M,R')
    resolveTypeWrite(M,R')
    return not(top-level-local)
  if(R' is parameter)
    M' = callerOf(M)
    return findOriginOfRef(M',R')
```

Figure 7. Type resolving for data-write analysis

3. Finally, the algorithm analyses the return statements by resolving the type of the data returned.

The algorithm enters the *resolveTypeWrite* method for all of the method parameters, class members and returned data to handle the different cases of feature types adequately (Figure 7):

- *Primitive type:* if this feature is identified as member or returned, all assignments to it within the method are annotated as data-write access (*primDataWrite*). However, it is important to find the source of this data. If the data origin is a parameter of a top-level method, no data-write access is necessary.

- *Array type:* if the feature is identified as member or returned, all assignments to it are annotated as simple reference-type data-write accesses. Moreover, the accesses to its origin are also to be recursively analysed (*resolveArrTypeWrite*).

  Assignment to an array element is annotated as potential data-write access to the array structure (*ArrayDataWrite*). Later, the algorithm analyses all potential accesses to the origin of the array-element assignment (*resolveTypeRead* and *resolveTypeWrite* for array element *E*).

- *Reference type:* if this feature is identified as member or returned, all assignments to it are annotated as potential data-write access (*RefDataWrite*). However, we have to find the origin of the reference type data (*findOriginOfRef*). Once the program representation is traversed in a backward fashion and the origin is marked as *top-level-local* (local within the scope of top-level method).

  The backward traversal (Figure 7,*findOriginOfRef*) recursively searches for any potential origin of the data assignment. If the data reference is local to given method, a data-write access is annotated. In the case that the data reference origin is a class member the algorithm starts forward traversal to resolve all read and/or write accesses on the reference origin. Finally, if the reference origin is identified as a method parameter a new backward traversal is started in the caller to this method. Thus, all possible cases are handled.

The backward analysis, as the forward analysis, adds annotations to particular code features, while traversing the cloned sub-program representation. The annotations are used in the instrumentation phase.

4.3. Instrumentation

The instrumentation phase of the data-access analysis modifies the program representation with profiler specific code based on annotations from the previous phases. It uses the instrumentation technique presented in [6]. Currently, the instrumentation processes the code for accesses to Java arrays and primitive-type scalars resulting in an appropriate collection of data-access information in the profiler. The annotations for primitive-type read and write accesses are transformed into the actual profiler-specific code for primitive-type data-access profiling, using data-access counter number 0 (*drd(0), dwr(0)*).

The annotations for array-type data uniquely identify each array within the thread-local scope, inserting unique data-read and data-write access counters for each thread and its accessed arrays separately. Moreover, the instrumenter adds information on the array index to the program representation, using *drd(ArrayID, ElementIndex) and dwr(ArrayID, ElementIndex)* calls to profiler, which allows generation of detailed array-access traces.

The annotations for reference-type accesses are handled depending on the situation in the program code, as explained in Section 4. The instrumenter inserts an appropriate code for the explicit data copying of all data members of given object (Figure 3-b)).

data-access	
prof.drd(0)	read access to primitive type
prof.drd(id,idx)	read access to array id index idx
prof.dwr(0)	write access to primitive type
prof.dwr(id,idx)	write access to array id index idx

Table 1. Profiler API: extension for data-access analysis

5. Run-time profiling support

5.1. Interface to the profiler

All the features described above are made available to the designer at Java source-code level via the Profiler API. The features for parallel program performance analysis were presented in our previous approach([18], ExtAPI). We extend the profiler interface with the data-access support (Table 1). The Profiler API is implemented completely in Java.

5.2. Parallel java profiler

The parallel profiler is the essential element of the performance analysis framework. It implements the concept of parallel-execution time [18]. Moreover, we extend that functionality with the above described support for data-access profiling.

The profiler uses a modified implementation of the JVM to execute the profiled program. Each task/thread of the program is assigned a separate timer and an unique identification number to correctly propagate the execution-time information of the simulated parallel execution [18]. The profiler executes the program and collects information on program execution and accesses to shared-data section. The information on shared-data accesses is stored in a special data-access counters. Thus, each separate thread has its own set of these counters.

If a detailed report mode is specified, the profiler produces complete program execution trace (Figure 8). It contains information about synchronisation between different threads, methods called within the threads and data read and write accesses. For each of these events the profiler reports the current parallel-execution time stamp and type of operation performed. In case of data-access operation, it reports type of data accesses and access type. If an array element is accessed, the array identification number and array-element index are reported.

Compared to our previous work, we use Jikes Research Virtual Machine [2] for implementation of the profiler. The main reason for this change was bad performance of the previous interpreter.

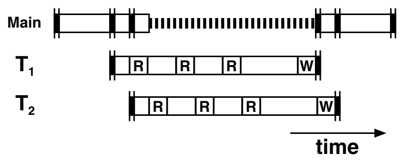

Figure 8. Program trace: the profiler generates program trace consisting of threads executing, operations on threads, synchronisation and data accesses to the shared data (Figure 2).

5.3. Parallel performance metrics

The post-processing phase of the performance analysis consist of a critical-path analysis algorithm presented in previous work [18] and data-access trace interpretation.

The data-access trace interpretation consists of annotation of different types of data accesses with a performance cost - in this case, data access time to the shared data. For annotation of shared data acccesses with realistic figures, we use the Micron SDRAM model [3]. The cost of random access to the shared data is 72ns, while the more optimistic page-mode access requires only 8ns. This approach can be further extended with data-reuse modelling shifting the trade-off towards more optimistic page-mode access. Moreover, a burst-mode access to the shared data can be introduced if the corresponding data-access trace has been identified.

6. Experimental results

In experiments accomplished we have focused on the usability of the proposed framework for an evaluation of different parallel alternatives of realistic programs. We use the host platform and corresponding memory model[3] to obtain the absolute timing information for program execution and data communication, while the main interest of our platform-independent program characterisation is in the relative comparison between them.

6.1. Experiments analysing task-creation overhead

For the task-creation overhead we have performed a number of simple tests. The results obtained show that the task-creation overhead ranges from 85μs to 725μs. We conclude that the task creation overhead for our model is negligible if the actual task-execution time is significantly bigger than the average task-creation time (331 μs). It will be seen in the next subsection that a typical task-execution time, for embedded multimedia applications, is in the range of tens to hundreds of milliseconds.

th	t[ms]	t_{rnd}[ms]	t_{pg}[ms]	id	#rd	#wr
1	3318	-	-	-	-	-
4	1516	-	-	x,5	-	-
		9.51	1.06	0,8	132k	13
		21.4	2.38	1,8	131k	166k
		9.48	1.05	0,9	132k	15
		14.3	1.58	1,9	-	198k
		0.60	0.07	0,10	8260	7
		1.31	0.15	1,10	-	18k
		9.48	1.05	0,11	132k	121
		17.5	1.94	1,11	-	242k

Table 2. Results for 3D program

th	t[ms]	t_{rnd}[ms]	t_{pg}[ms]	id	#rd	#wr
1	36927	-		-	-	-
2	18717	2870	319	0,1	24.7M	15.2M
		79.9	8.88	1,1	1.1M	-
		8.50	0.94	2,1	-	118k
		398	44.2	3,1	5.5M	-
		8.49	0.94	7,1	-	117k
		43.9	4.88	9,1	610k	-
		81.8	9.08	11,1	568k	568k
		1439	160	12,1	2.0M	27.1k
		3028	336	13,1	2.1M	40M
		49.0	5.45	47,1	-	681k
		637	70.8	48,1	-	8.9M

Table 3. Results for MPEG program

6.2. Experiments exploring program performance and data communication metrics

For the evaluation of the performance analysis framework we have used a 3D application [15], an MPEG video player [14] and the following set of applications from the Java Grande Forum Thread Benchmark Suite [16]: JGFCrypt, JGFSparseMatMult, JGFRayTracer and JGFMonteCarlo.

In case of the 3D application, we have used scenario (local-view texture decoding) that takes relatively short time and shows the typical program behaviour. The communication (Table 2) between the threads and the shared data region is relatively low (approx. 6% of total execution time).

th	t[ms]	t_{rnd}[ms]	t_{pg}[ms]	id	#rd	#wr
1	5060	0	0	0,5	8	-
		432	48	1,4,5	6M	-
		2808	312	2,5	39M	-
2	2775	0	0	0,x	8	-
		216	24	1,4,x	3M	-
		1404	156	2,x	19.5M	-
4	1624	0	0	0,x	8	-
		108	12	1,4,x	1.5M	-
		702	78	2,x	9.75M	-

Table 4. Results for JGFCrypt

th	t[ms]	t_{rnd}[ms]	t_{pg}[ms]	id	#rd	#wr
1	20071	3600	400	0,3-7,5	50M	-
		7200	800	2,5	50M	50M
2	10147	3600	400	0,3-7,x	25M	-
		3602	400	2,x	25M	25M
4	5207	901	100	0,3-7,x	12.5M	-
		1802	200	2,x	12.5M	12.5M

Table 5. Results for JGFSparseMatMult

th	t[ms]	t_{rnd}[ms]	t_{pg}[ms]	id	#rd	#wr
1	108648	8984	998	0,5	85.6M	39.2M
		7502	834	1,5	104M	-
		159	18	2,5	2.21M	-
2	54466	4489	499	0,x	42.8M	19.6M
		3752	17	1,x	52.1M	-
		79	9	2,x	1.10M	-
4	27398	2238	249	0,x	21.3M	9.76M
		1868	208	1,x	25.9M	-
		40	4	2,x	550k	-

Table 6. Results for JGFRayTracer

We have profiled two scenarios: the first scenario is the original single-threaded program. Its execution time is approximately 3.3 seconds. The second scenario is a parallel version of the same program using four threads. The resulting execution time (without communication overhead) is approximately 1.5 seconds. The amount of data accesses is in the last two columns (namely $\#rd$ and $\#wr$). For the calculation of communication time we use two data-access modes: random-access mode $T_{rnd}[ms]$: data-access takes 72ns, and page-access mode $T_{pg}[ms]$: data-access takes 8ns. The array index (id) consists of thread- and counter-identification number. Except for counter zero, the counters with higher identification numbers correspond to particular arrays in the program.

Finally, we interpret the results of the performance analysis(Table 8): we calculate program communication time (T_C) in two ways. Either the data accesses to the shared data are fully overlapping ($3D_{ca}$ - concurrent access) or the shared-data accesses are sequential ($3D_{sa}$ - sequential access). The total execution time (T_T) is a sum of execution time and data-communication time. Thus, the achievable speed-up ranges from the speedup for random data-

th	t[ms]	t_{rnd}[ms]	t_{pg}[ms]	id	#rd	#wr
1	50355	18.7	2.08	0,5	90k	170k
		720	80	2,3,5	-	10M
2	27278	9.36	1.04	0,x	45k	85k
		360	40	2,3,x	-	5M
4	16152	4.68	0.52	0,x	22.5k	42.5k
		180	20	2,3,x	-	2.5M

Table 7. Results for JGFMonteCarlo

program	$T_{C,pg}[ms]$	$T_{C,rnd}[ms]$	$T_{T,pg}[ms]$	$T_{T,rnd}[ms]$	S_0	S_{pg}	S_{rnd}
$3D_{4ca}$	3.44	30.9	1519	1546	2.18	2.18	2.15
$3D_{4sa}$	9.28	83.6	1525	1600	2.18	2.18	2.07
$MPEG_2$	962	8653	19679	27370	1.97	1.88	1.35
$Crypt_2$	204	1836	2979	4611	1.82	1.70	1.10
$Crypt_{4ca}$	102	918	1726	2542	3.12	2.93	1.99
$Crypt_{4sa}$	306	2754	1930	4378	3.12	2.62	1.16
$Matrix_2$	1800	1980	11947	29954	1.98	1.68	0.67
$Matrix_{4ca}$	800	7208	6007	12415	3.85	3.34	1.62
$Matrix_{4sa}$	2400	21624	7607	26831	3.85	2.63	0.75
$RayT_2$	925	8320	55391	62786	1.99	1.96	1.73
$RayT_{4ca}$	465	4183	27863	31581	3.97	3.90	3.44
$RayT_{4sa}$	1385	12466	28783	39864	3.97	3.77	2.73
$MCarlo_2$	81	729	27359	28007	1.85	1.84	1.80
$MCarlo_{4ca}$	40.5	365	16193	16517	3.12	3.11	3.05
$MCarlo_{4sa}$	120	1095	16272	17247	3.12	3.09	2.92

Table 8. Interpreted analysis results: $program_{Xca/sa}$ (X - number of threads, ca/sa - concurrent or sequential data-access mode), communication time, total-execution time and speedup for zero/page/random-mode communication.

access model (S_{rnd}) to the speedup for page-mode data-access model (S_{pg}). For comparison reasons we present also speedup for an ideal reference (S_0 - no data-communication overhead).

The MPEG video player is an application which uses a separate thread for execution of the video decoder. The results (Table 3) show that it is a heavily data-dominated application. The analysis identifies a number of arrays intensively accessed by the video-decoding thread. Based on the interpretation of the analysis results (Table 6.2, $MPEG_2$), we see that in case of random data-access mode, the communication corresponds to 31% of the total execution time. Thus, compared to the ideal reference (S_0 = 1.97), the speed-up is considerably degraded (S_{rnd} = 1.35).

We have analysed two programs (JGFCrypt and JGFSparseMatMult) from section 2 and two other (JGFRayTracer and JGFMonteCarlo) from section 3 of the Java Grande Forum Thread Benchmarks, using single-thread, two-thread and four-thread configuration. The JGFCrypt benchmark program is a data-dominated application (Table 4), e.g., in four-thread configuration each thread performs approximately 10 million data-read accesses and 1.5 million data-write accesses. Thus, the overall performance of the program is heavily dependent on the data-access mode (Table 6.2): the achievable speedup, in the best case scenario (S_{pg} at $Crypt_4ca$ - concurrent, page-mode data access), is approximately 2.93, while it drops down to 1.16 for the worst case scenario (S_{rnd} at $Crypt_4sa$ - sequential, random-mode data access). As can be seen from the results, the JGFSparseMatMult benchmark programs is another data-dominated application. Moreover, the JGFSparseMatMult worst-case scenario (Table 6.2, S_M at $Matrix_2$) is 33% slower then the sequential execution. The JGFRayTracer and JGFMonteCarlo benchmark programs

```
#### DAA start
#### methods + sparsematmult.SparseRunner.run
#DAA ENTERING sparsematmult.SparseRunner.run
#DAA this invoke @ run
...
#DAA going params @ run
...
#DAA going members @ run
...
#DAA EXITING sparsematmult.SparseRunner.run
#DAA #counters: rd = 10
               wr = 1
#DAA #methods = 1
#DAA time: 245 [ms]
```

Figure 9. A fraction of the report from the design-time data-access analysis tool.

are more complex applications yet less data-dominated. The JGFRayTracer communication time ranges from 1.7% (S_{pg} at $RayT_2$ or $RayT_4ca$) to 31.3% (S_{rnd} at $RayT_4sa$)) while it is only 0.3 - 6.3% for the JGFMonteCarlo program. Thus, the final speedup for those programs does not depend on the data-communication as heavily as in the previous examples.

6.3. Evaluation of the analysis tools

We have evaluated the usability of our tool from designer's point of view and computational complexity of the design-time algorithms. The current user interface is batch-based and the only required input from designer is a list of threads/methods to analyse. The tool automatically analyses the data-accesses, instruments the code, profiles the program and generates reports. The design-time data-access analysis tool generates a report on assignment of method

```
============ statistics =============
VM thd[8]   = 4477709 type java.lang.Thread
VM cnt[8,1] = 4430072
VM drd[8,0] = 12495801
VM drd[8,1] = 200
VM drd[8,7] = 12495400
VM main[5]  = 5207051
VM cnt[5,1] = 5141155
VM drd[5,0] = 12459201
VM drd[5,7] = 12458800
VM drd[5,2] = 12458600
...
```

Figure 10. An example report on program execution (JGFSparseMatMult).

program	T_{trans}[ms]	#methods	#rd	#wr
3D	3819	63	43	133
MPEG	33798	41	436	337
Crypt	1266	2	22	8
MatMulti	245	2	10	1
RayTracer	4927	29	134	66
Montecarlo	1137	23	17	19

Table 9. Data-access analysis: program, transformation time, number of sub-program methods, identified read and write accesses.

timers to the set of sub-program methods, assignment of data-access counters to particular shared data elements (Figure 9)). An example of the generated profiling report is shown in Figure 10). The tool also provides statistics on its execution, including transformation time, number of analysed methods, number of identified potential read and write accesses (Table 9).

The upper bound of computation complexity of the data-access analysis algorithms is $n \times m_R \times k_R$ for forward pass and $n \times m_W \times k_W$ for backward pass, where n is number of methods selected by designer, m_R (m_W) is number of methods access from selected methods for data read (resp. write) and k_R (k_W) is number of shared data which are read (resp. written) within the scope of selected methods. However, we have not observed this computational complexity in behaviour of the tool for tested programs.

The performance and data-access analysis framework allows analysis of complex Java applications, providing detailed insight into their performance and data-access characteristics.

7. Conclusions

We have introduced the performance and data-access analysis part of a transformation framework for exploration of task-level parallelism in sequential object-oriented programs. The main difference of our approach compared to related work is the introduction of a performance-analysis technique that implements the concept of parallel execution time. We have extended this approach with shared-data model for data communication between the program tasks. This allows one to simulate the behaviour of the parallel program with respect to the architectural constraints of the target platform. To increase efficiency of profiling, we have implemented automatic performance and data-access analysis and instrumentation of Java programs.

We have demonstrated the potential of our technique on a number of realistic test applications, showing its suitability for exploration and analysis of the parallel program performance on the target multi-processor platforms.

References

[1] Vallee-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java Optimization Framework, Proc. of CASCON, 1999

[2] Jikes Research Virtual Machine: http://www-124.ibm.com/developerworks/oss/jikesrvm/

[3] Micron: Calculating Memory System Power For DDR, www.micron.com, TN-46-03

[4] Ding, C., Zhong, Y.: Compiler-Directed Run-time Monitoring of Program Data Access, Proceedings of the workshop on Memory system performance, Berlin, Germany, pp.1-12, 2003

[5] Bormans, J., Denolf, K., Wuytack, S., Nachtergaele L. and Bolsens, I.: Integrating System-Level Low Power Methodologies into a Real-Life Design Flow, Proceeding of IEEE Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS), Kos, Greece, pp.19-28, 1999

[6] Leeman, M. et al.: Power Estimation Approach of Dynamic Data Storage on a Hardware Software Boundary Level, Proceeding of IEEE Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS), Torino, Italy, pp.289-298, 2003

[7] Chakrabarti, S., Gupta, M., Choi, J.D: Global Communication Analysis and Optimisation, Proceedings of Conference on Programming Language Design and Implementation, pp.68-78, 1996

[8] Tseng, C-W.: Communication Analysis for Shared and Distributed Memory Machines, Proceedings of the Workshop on Compiler Optimizations on Distributed Memory Systems, 1995

[9] Mellor-Crummey, J., Fowler, R., Whalley, D.: Tools for Application-Oriented Performance Tuning, Technical Report TR01-375, Rice University, 2001

[10] Adve, V.S. and Vernon, M.K.: A Deterministic Model for Parallel Program Performance Evaluation, Technical Report TR98-333, Rice University, 1998

[11] Vetter, J.: Performance Analysis of Distributed Applications using Automatic Classification of Communication Inefficiencies, Proceeding of ACM International Conference on Supercomputing, Santa Fe, USA, 2000

[12] Miller, B.P., et al.: The Paradyn Parallel Performance Measurement Tool, Journal IEEE Computer, vol.28, num.11, pp.27-46, 1995

[13] Haake, B., Schauser, K.E., Scheiman, C.: Profiling a parallel language based on fine-grained communication, Proceedings of the ACM/IEEE conference on Supercomputing, Pittsburgh, USA, 1996

[14] Anders, J.: MPEG-1 player in Java, http://rnvs.informatik.tu-chemnitz.de/ jan/MPEG/MPEG_Play.html

[15] Walser, P.: IDX 3D engine, http://www2.active.ch/ proxima

[16] Java Grande Forum Benchmarks, http://www.epcc.ed.ac.uk/javagrande/javag.html

[17] Gosling, J., Joy, B., Steele, G. and Bracha, G.: The Java Language Specification, Second Edition Addison-Wesley, 2000

[18] R.Stahl et al.: Performance Analysis for Identification of (Sub)task-Level Parallelism in Java, Proceedings of SCOPES'03, Austria, 2003