Design-Time Data-Access Analysis for Parallel Java Programs with Shared-Memory Communication Model

R. Stahl, 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 architectures 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 estimate the potential parallelism from sequential object-oriented programs in Java.
Data-access analysis is one of the crucial techniques for estimation of the transformation effects. We have implemented support for platform-independent data-access analysis and profiling of Java programs. Herein, we focus on the technique for design-time data-access analysis. It complements our earlier work on parallel performance analysis. We demonstrate the feasibility and effectiveness of our approach on a number of Java applications.

1 Context and Related Work

Data-access and communication analysis for parallel programs is an important topic, as motivated in the abstract. 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. 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 way: first, we have introduced the concept of parallel execution [15], 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 [16] for which we analyse the computation as well as communication cost.

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 206–213, 2004.

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. Miller et al. [10] 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. Haake et al. [11] 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 offline. Another approach, implemented by Vetter [9] 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.

2 Parallel-Performance Analysis Framework for Java

The proposed performance analysis tool is based on a concept of parallel-execution time [15, 16], 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. The tools work as follows. Firstly, the program is automatically transformed based on designer's input constraints. This phase consists of two complementary transformations: parallel performance analysis and data-access analysis. 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.

Herein, we focus on the design-time data-access analysis. The analysis identifies all potential data-accesses which are then profiled at run time. For this purpose, we define a data-access model (Figure 1) that consists of main execution thread, number of separate threads and shared data. The implementation of the transformation passes as well as intra and inter-procedural program analysis is based on the existing transformations in the SOOT optimisation framework [1].

3 Design-Time Data-Access Analysis

The data-access model (Figure 1) is used as a representation for modelling the accesses to the data shared between different program tasks present in a sequential or parallel program. 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/methods and shared data. No cache memories are included yet this extention is possible. All shared data belong to main method and they are stored in the shared-data section. The separate methods require an amount of data to be read from the shared-data section before they can proceed with execution. On the other hand, the methods generate and write an amount of data to be stored back to the shared-data section.

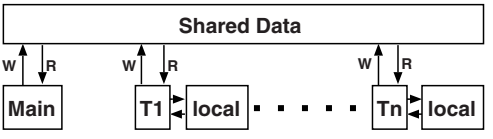

Fig. 1. Data-access model: a shared memory communication model of an abstract architecture, used in the data-access analysis to separate local and non-local accesses of separate threads T_i.

The designer specifies a set of separate methods, and the analysis identifies all the accesses performed on the shared data. To achieve this, it traverses the corresponding parts of the program representation based on a method-call graph (Figure 2). In the case of a polymorphic method call the analysis resolves all method call candidates while only one of them is selected and profiled at run-time. The analysis identifies all the data created outside the scope of a separate method, and accessed within its scope. Based on the Java programming language specification [2], access to this shared data can be performed only via method parameters (P_M - list of parameter for method M), class members (F_M - list of class fields) and return statements (R_M). We can conceptually split the data-access analysis into two parts: data-read and data-write analysis:

$\forall M \in SeparateMethods \rightarrow analyseR(M); analyseW(M)$

The data-access analysis algorithms are implemented recursively because of their iterative nature without manifest bounds on the exploration depth. The upper bound of computation complexity of the algorithms is $O(n \times m_R \times k_R)$ for forward pass and $O(n \times m_W \times k_W)$ for backward pass, where n is number of methods selected by designer, m_R (m_W) is number of methods accessed 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 worst-case behaviour in analysis of tested programs.

The data-access analysis identifies and distinguishes three groups of data types:

- primitive type $T_P = char, short, integer, long, float, double$,
- reference type $T_R = ObjectReference, ArrayReference$,
- array type $T_A = ArrayReference$ [1].

[1] The Java arrays implement distinct concept within the Java language so we consider them a special case of the reference type.

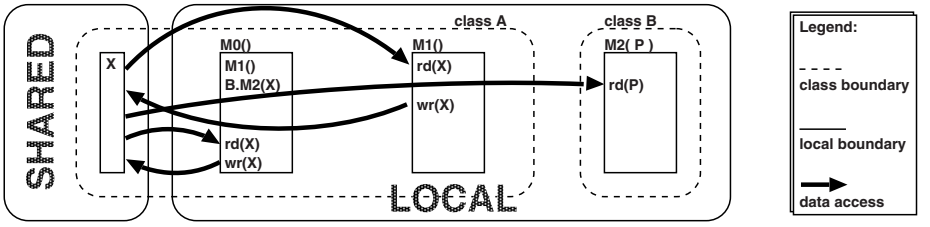

Fig. 2. Example: the data-access analysis identifies all possible read and write accesses from thread's local scope to shared-data section, where M0() is the separate (or thread's main) method.

3.1 Forward Data-Read Analysis

The data-read analysis identifies the read accesses to the shared data resulting in purely forward recursive traversal of the program representation. The potential candidates for the data-read access are only method parameters and class members. The analysis consists of the following steps: analysis of method calls for own class methods ($ClassMethods$), analysis of method parameters (P_M) and analysis of class members (F_M) accessed within each separate method.

$$
\begin{aligned}
&ana\ lyseR(M):\\
&\ if\ \ (M \notin SeparateMethods) \rightarrow M_c = clone(M)\\
&\ \forall\ \ M' : isInvokedIn(M, M') \wedge isClassMethods(M, M') \rightarrow analyseR(M')\\
&\ \forall\ \ P_M : isReadIn(M, P_M) \rightarrow analyseTypeR(P_M)\\
&\ \forall\ \ F_M, isReadIn(M, F_M) \rightarrow analyseTypeR(F_M); FieldList.identify(F_M, M_c)
\end{aligned}
$$

Method $clone(M)$ creates a clone M_c of given method M with unique thread identification tag. Each top-level method and sub-graph of the method-call graph accessed from this method is identified by an identification tag. Method M' is analysed if it is invoked with M method body ($isInvokedIn(M, M')$) and it is a method implemented with the same class ($M' \in ClassMethods$).

Method $FieldList.identify(Field, Method)$ identifies and resolves given class member within global program scope. The analysis uses this global information to resolve accesses to all members of given classes. Thus, it removes aliasing of the members which is introduces by assignments in different methods of the program. This alias removal results in minimal and realistic data-access patterns. As shown below, the same method is used for the data-write analysis that results in accurate information on read and write accesses to all the class members.

The algorithm enters $analyseTypeR(P_M)$ method for each of the method parameters P_M and class members F_M used within M method body ($isReadIn(M, P_M)$). The method analyses the given argument based on its type (primitive T_P, reference T_R or array T_A) as follows.

$$
\begin{aligned}
&ana\ lyse\ TypeR(P_M):\\
&\ if\ (ass\ ignedTo(P'_M, P_M)) \rightarrow analyseTypeR(P'_M)\\
&\ if\ (P_M \in T_P) \rightarrow addRdType(M_c, P_M, T_P)
\end{aligned}
$$

$$
\begin{aligned}
&if\ (P_M \in T_R) \rightarrow addRdType(M_c, P_M, T_R)\\
&\quad \forall\ \ M' : isClassMethods(P_M, M') \wedge isInvokedIn(M, M') \rightarrow analyseR(M')\\
&if\ (P_M \in T_A) \rightarrow \forall E_M = P_M[idx] : E_M.isReadIn(M) \rightarrow analyseTypeR(E_M);\\
&\qquad addRdType(M_c, E_M, T_A); ArrayList.identify(P_M, M_c);
\end{aligned}
$$

Call to $addRdType(Method, Parameter, Type)$ does the actual code annotation for later use in other phases of program transformation. The method adds parameter-specific annotation to the code segment of cloned method (M_c). The annotation consists of parameter name (P_M) and type (T_P, T_R or T_A). Each data (parameter or member) is assigned a unique global identification tag that it is identifiable within the global program scope, i.e., for all the separate methods.

In the case of array data type (T_A), the analysis annotates all array elements (E_M) accessed withing the method body ($isReadIn(M, E_M)$). Thus, the annotation made by method $addRdType(M_c, E_M, T_A)$ includes also the actual array index, e.g., if the annotated program is later profiled the actual index value is identified, which results in a detailed data-access trace for the arrays.

Method $ArrayList.identify(Array, Method)$ has similar usage as $FieldList$ method yet it resolves even method-local array data. This way, the analysis keeps global information on the accesses to individual array elements that results in alias removal on the level of array elements.

3.2 Backward Data-Write Analysis

The data-write analysis traverses the representation forward until it finds any write access to a shared data then it starts a backward traversal to identify the origin of the data assignment. Furthermore, it inspects all potential accesses to this data. Thus, it uses the forward analysis to resolve all read accesses to this newly identified data which is eventually written into shared-data section. The potential candidates are non-primitive method parameters, class members, and return statements.

$$
\begin{aligned}
&ana\ lyseW(M):\\
&if\ \ (M \notin SeparateMethods) \rightarrow M_c = clone(M)\\
&\forall\ \ M' : isInvokedIn(M, M') \wedge M' \in ClassMethods \rightarrow analyseW(M')\\
&\forall\ \ P_M : isWrittenIn(M, P_M) \rightarrow analyseTypeW(P_M)\\
&\forall\ \ F_M : isWrittenIn(M, F_M) \rightarrow analyseTypeW(F_M); FieldList.identify(F_M, M_c)\\
&if\ \ (hasReturn(M)) \rightarrow R_M = returnedFrom(M); analyseTypeW(R_M)
\end{aligned}
$$

The method $analyseTypeW(P_M)$ is defined as follows.

$$
\begin{aligned}
&ana\ lyse\ TypeW(P_M):\\
&if\ (P_M \in T_P) \rightarrow DataWriteType(M_c, P_M, T_P)\\
&if\ (P_M \in T_R) \rightarrow\\
&\quad \forall\ \ M' : isClassMethods(P_M, M') \wedge isInvokedIn(M, M') \rightarrow analyseW(M')\\
&\quad if\ \ (assignedFrom(P'_M, P_M) \wedge findOrigin(P'_M)) \rightarrow addWrType(M_c, P_M, T_R)\\
&if\ (P_M \in T_A) \rightarrow
\end{aligned}
$$

$$
\begin{aligned}
&if \quad (assignedFrom(P'_M, P_M) \wedge findOrigin(P'_M)) \rightarrow addWrType(M_c, P_M, T_A)\\
&\forall \quad E_M = P_M[idx] : asignedFrom(E'_M, E_M) \wedge findOrigin(E'_M) :\\
&\quad addWrType(M_c, E_M, T_A); ArrayList.identify(P_M, M_c);\\
&\quad analyseTypeR(E_M); analyseTypeW(E_M)
\end{aligned}
$$

Method $findOrigin(P_M)$ is called for each assignment ($assignedFrom(P'_M, P_M)$) of a reference, array and array element . Based in the result of its analysis an appropriate method annotation is made ($addWrType(M_c, P_M, Type)$).

$$
\begin{aligned}
&fin\ dOrigin(P_M) :\\
&if \quad (isDefLocally(P_M)) \rightarrow return(true)\\
&if \quad (isDefByMember(P_M) \wedge assignedFrom(P'_M, P_M)) \rightarrow\\
&\quad analyseTypeR(P'_M); analyseTypeW(P_M); return(false)\\
&if \quad (isDefByParameter(P_M)) \rightarrow\\
&\quad if(M \in SeparateMethods) \rightarrow return(false)\\
&\quad M' = getCaller(M); P'_M = getParam(P_M, M'); return(findOrigin(P'_M))
\end{aligned}
$$

Method $findOrigin(P_M)$ traverses the program representation backwards to search for the original assignment of the given data (P_M). It returns true only in case of true local assignment, i.e., if the given data was created locally within the scope of given separate method.

In the case of definition by parameter ($isDefByParameter(P_M)$), if the analysis operates in the representation of a particular separate method ($SeparateMethods$) the returned value is false and no further analysis is needed. This situation corresponds to assignment from input parameter of the given separate method to its output data, which means that no actual read and/or write access is performed.

Design-time data-access analysis traverses all potential method calls in the subgraph of the method-call graph while performing alias checks to isolate given data structures and thus provide accurate information on potential data accesses in the program.

4 Experimental Results

In experiments accomplished we have focused on the usability of the proposed framework [16]. We have used 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. The data communication timing is calculated as a product of the number of memory accesses and data-access timing for the above referenced memory model.

For the evaluation of the performance analysis framework we have used a 3D application [13], an MPEG video player [12] and the following set of applications from the Java Grande Forum Thread Benchmark Suite [14]: JGFCrypt, JGFSparseMatMult,

Table 1. 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.

program	$T_{C,pg}[ms]$	$T_{C,rnd}[ms]$	$T_{T,pg}[ms]$	$T_{T,rnd}[ms]$	$\mathbf{S_0}$	$\mathbf{S_{pg}}$	$\mathbf{S_{rnd}}$
$3D_{4ca}$	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**

JGFRayTracer and JGFMonteCarlo. The results of the performance analysis can be interpreted as follows (Table 1): data-communication time (T_C) is calculated separately for concurrent and sequential data accesses to the shared-data section ($program_{Xca/sa}$). 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-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).

An example of application with heavy data communication is the MPEG video player. Based on the interpretation of the analysis results (Table 1, $MPEG_2$), 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 realistic speed-up is considerably degraded (S_{rnd} = 1.35). Another example of such an application is JGFSpareMatMult (Table 1, $Matrix_4$). On the other hand, an example of less data dominated application is the JGFMonteCarlo benchmark programs. Even though, the program complexity is similar the communication time ranges from 0.3 - 6.3%. Thus, the final speedup (ranging from 2.92 to 3.11, with reference speedup of 3.12) does not depend on the data-communication as heavily as in the previous example.

The execution time of the data-access analysis tool ranges from 245 ms for JGFSparseMatMult program (2 analysed methods and 11 identified data accesses) to 33798 ms for MPEG video player (41 methods and 773 data accesses), executed using Sun J2SDK1.3.1 on desktop PC with Pentium-4 1.6GHz, 640MB RAM. Typical execution time is in the range of 1 to 4 seconds (programs with 20 to 60 methods and 10 to 140 identified accesses).

5 Conclusions

We have introduced the design-time data-access analysis which is a crucial part of our 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 the design-time data-access analysis for the programs with shared-memory communication model. To increase the usability of our technique we have implemented automatic tool that performs the data-access analysis on Java programs.

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. Gosling, J., Joy, B., Steele, G. and Bracha, G.: The Java Language Specification, Second Edition Addison-Wesley, 2000
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. Vetter, J.: Performance Analysis of Distributed Applications using Automatic Classification of Communication Inefficiencies, Proceeding of ACM International Conference on Supercomputing, Santa Fe, USA, 2000
10. Miller, B.P., et al.: The Paradyn Parallel Performance Measurement Tool, Journal IEEE Computer, vol.28, num.11, pp.27-46, 1995
11. 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
12. Anders, J.: MPEG-1 player in Java, http://rnvs.informatik.tu-chemnitz.de/ jan/MPEG/MPEG_Play.html
13. Walser, P.: IDX 3D engine, http://www2.active.ch/ proxima
14. Java Grande Forum Benchmarks, http://www.epcc.ed.ac.uk/javagrande/javag.html
15. R.Stahl et al.: Performance Analysis for Identification of (Sub)task-Level Parallelism in Java, Proceedings of SCOPES'03, Austria, 2003
16. R.Stahl et al.: High-Level Data-Access Analysis for Characterisation of (Sub)task-Level Parallelism in Java, to be published in Proceedings of HIPS'04 workshop, Santa Fe, USA, April 2004