Embla – Data Dependence Profiling for Parallel Programming

Karl-Filip Faxén, Konstantin Popov, Sverker Janson
Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, Sweden
{kff,kost,sverker}@sics.se

Abstract

With the proliferation of multicore processors, there is an urgent need for tools and methodologies supporting parallelization of existing applications. In this paper we present a novel tool for aiding programmers in parallelizing programs. The tool, Embla, is based on the Valgrind framework, and allows the user to discover the data dependences in a sequential program, thereby exposing opportunities for parallelization. Embla performs a dynamic analysis, and records dependences as they arise during program execution. It reports an optimistic view of parallelizable sequences, and ignores dependences that do not arise during execution. Moreover, since the tool instruments the machine code of the program, it is largely language independent.

We also investigate the relation between the dependencies observed for different inputs to the same program and present an analysis of the SPEC CPU2006 benchmark 403.gcc.

1 Introduction

Parallel programming is no longer optional. To enjoy continued performance gains with future generation multicore processors, application developers must parallelize all software, old and new [12, 9, 8, 13]. For scalable parallel performance, program execution must be divided into large numbers of independent tasks that can be scheduled on available cores and hardware threads by runtime systems. For some classes of programs, static analysis and automatic parallelization is feasible [5], but with the current state-of-the-art, most software requires manual parallelization. Our work aims to help developers find the potential for parallelism in programs, in particular by providing efficient tool support. In this paper, we present a data dependence profiling approach to the parallelization problem, an efficient algorithm to project data dependences onto relevant parts of the program code, and its implementation, the tool Embla, as well as an analysis of the dependencies observed when running the GCC compiler on different inputs.

```
p();          spawn p();
q();          q();
r();          sync;
              r();
```

Figure 1. Example of Fork/join parallelism.

We are interested in the following methodology for constructing parallel programs: Start from a sequential program, identify independent parts of that program (here Embla can be used) and rewrite the program to obtain parallel executioon of the independent parts.

We will focus on parallelization by introducing fork-join parallelism. The fork-join framework was first introduced by Conway [3] and is used in many parallel programming environments, including Cilk [1], the Java fork/join framework [6], OpenMP [4], and Filaments [7].

Consider the program fragment in Figure 1 (left): Suppose that the calls to `p()` and `q()` are independent, but that the call to `r()` depends on the earlier calls. Then the call to `p()` can be executed in parallel with the call to `q()`, as shown to the right. Here we assume the availability of a construct `spawn` to start the call in parallel and `sync` to wait for all `spawn`'d activities to terminate (cf. [1]).

Embla can help programmers find independent parts of the program code. The availability of such independent program parts depends on the algorithms used and can be further limited by sequential programming artifacts, such as re-use of variables and sequential book-keeping in an otherwise parallelizable algorithm. Data dependence information can potentially help identify and remove such obstacles to parallel execution, but this will not be further discussed here.

Parallelizing compilers mostly target loop parallelization based on static data dependence analysis methods [5]. Such analyzers are by necessity conservative, and use approximations that are always safe. Analyzing more general code,

e.g., with pointers, remains a major challenge and correctness is typically guaranteed only in the absence of bounds violations. Consequently, it has proved difficult to parallelize programs automatically, and most production codes are either written in an explicitly parallel way or rely on speculative, run-time parallelization techniques [10, 2].

In contrast, Embla observes the actual data dependences that occur during program execution, projects them onto relevant program parts, and interprets the lack of a runtime data dependence as an indication that the program parts involved are likely to be independent. Developers will be responsible for selecting program inputs that generate representative program executions with good coverage.

By construction, the methodology mentioned above preserves the semantics and determinancy of the sequential program under the assumption that all dependencies are found. Should a dependence remain undetected, it might manifest itself as a difference in behavior between the parallel and sequential versions of the program for some input. Given that the sequential program is deterministic, rerunning it under Embla with the offending input will yield the missing dependence. Thus the programmining methodology supported by Embla replaces the problem of finding synchronization errors in the parallelized, nondeterministic, program with the problem of finding all of the relevant dependencies in the sequential program.

In addition to data dependences, Embla can be extended to deal with I/O dependences that limit parallelism. Another potential extension is to measure the potential speed improvement for parallelizing independent program parts and produce a report with suggested program transforms that will yield the maximum benefit. These extensions are both future work. This paper presents the mechanism for collecting runtime data dependence information.

2 Using Embla

To get a feeling for what dependence profiling is and what Embla can do, let us turn to the (contrived) example program in Figure 2, where we see, from left to right, line numbers, data dependence arrows and source lines.

A *data dependence* is a pair of references, not both reads, to overlapping memory locations with no intervening write. We will refer to these references as the *endpoints* of the dependence. For instance, in the figure, there is an arrow from line 13 to line 14 corresponding to the assignment to `q` (the *early* endpoint) followed by its use as an argument in `inc(q)` (the *late* endpoint). Embla internally distinguishes between flow (RAW), anti (WAR) and output (WAW) dependences, but we do not make that distinction in this paper. Embla can be instructed to show dependence types, which can be useful for figuring out the reasons for individual dependences.

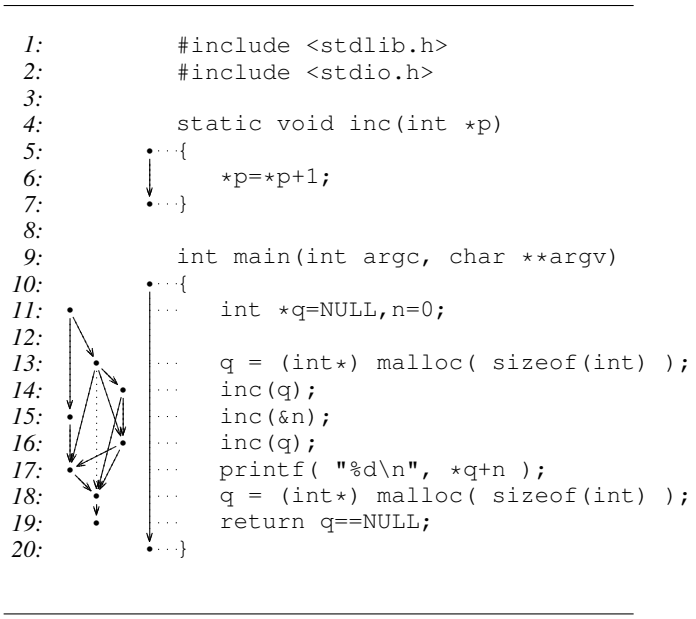

Figure 2. Example program with dependence graph

The endpoints of the dependence arrow discussed above are parts of the code for `main` itself, but Embla also tracks references made in function calls. For instance, there is a flow dependence from line 14 to line 16 representing the write in the first invocation of `inc` to the `malloc`'d area pointed to by `q` and the subsequent read of the same location by a later invocation of `inc`. These dependences are reported as pertaining to `main` rather than `inc`, although the endpoints are part of the latter function. But the importance of the dependence is that, in `main`, the calls on line 14 and 16 can not be made in parallel.

The dependence given with a dotted arrow (from line 13 to line 18) is due to manipulation of administrative data structures by `malloc`. If taken at face value such dependences will serialize all calls to `malloc`, but fortunately, the exact order of memory allocations is not important. If the parallelized version of the program uses a thread safe implementation of `malloc` these dependences are irrelevant and can be ignored. Embla maintains a suppression list, with functions that behave in this way. Similarly, dependences arise due to the manipulation of the call stack, which are also irrelevant for parallelization.

3 The Dependence Attribution Algorithm

We are interested in dependences between program statements, and in this section we discuss how to compute these dependences efficiently from the instruction level de-

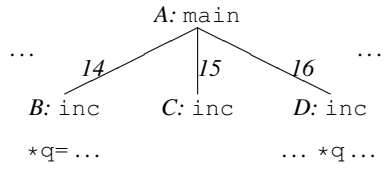

Figure 3. Part of the execution tree of Example 1, edges are annotated with the line number of the corresponding call.

pendences directly observed by Embla.

3.1 The execution tree

Consider the *instruction trace* S where each element corresponds to the execution of an instruction. From this trace we can construct a tree where the leaves are events from S, and each internal node is a call event followed by a sequence of nodes and a return event. We call this tree T the *execution tree* of S. Each node corresponds to some execution of a procedure body. Figure 3 shows part of the execution tree of Figure 2 where we have omitted the leaves. The nodes marked *B*, *C*, and *D* correspond to the three consecutive calls to `inc` at lines 14–16 of `main`.

The transformations we target will execute *siblings*, subtrees with the same parent, in parallel (in Figure 3, *B*, *C* and *D* are siblings). Hence, we are interested in dependences between siblings. These arise from the dependences in the instruction trace; if M and N are siblings and there is an instruction level dependence from an instruction in M to an instruction in N we have a (tree) dependence between M and N and we call M the *source* and N the *target* of the dependence.

In fact, each instruction level dependence yields exactly one dependence between siblings in the execution tree since there is only one node in the execution tree where the two events fall in two distinct children (siblings of each other) that could potentially be rearranged. We call that node the *nearest common ancestor* (NCA) of the endpoints of the instruction level dependence. Note that either or both of the children could be a leaf, in which case the corresponding dependence endpoint would be direct. For example, in Figure 3 the dependence between the write in *B* and the read in *D* yields only the tree dependence between the nodes *B* and *D* and the NCA of *B* and *D* is *A*.

The dependence is then reported as a dependence between the source lines associated with the source and target subtrees, respectively; in the example between lines 14 and 16 in `main`.

```
DependenceEdge( oldEvent, currEvent ) {
    oldLine = oldEvent.line;
    ncaNode = oldEvent.node;
    while( ncaNode is not on stack ) {
        oldLine = ncaNode.line;
        ncaNode = ncaNode.parent;
    }
    if( ncaNode != currEvent.node )
        currLine = ncaNode.next.line;
    else
        currLine = currEvent.line;
    return ( oldLine, currLine );
}
```

Figure 4. Constructing the dependence edge from the events corresponding to a previous and a new reference

3.2 Computing dependences

Embla uses two main data structures: The *trace pile*, which implements the execution tree, and the *memory table* which maps addresses to tree nodes corresponding to the last write and subsequent reads of that location. The trace pile contains the part of the execution tree corresponding to the part of the instruction trace that has been seen so far. For each reference, we look up the data address in the memory table. If the reference is a read, we use the previous write to generate a flow dependence (RAW). If it is a write, we use the previous write to construct an output dependence (WAW) as well as all reads since that write to construct anti dependences (WAR).

If there are several reads with no intervening write, a subsequent write (anti) depends on all of them. Since the reads do not depend on each other, we need to keep track of all of them in the memory table to generate the anti dependence edges explicitly. When that write has been processed the read list can be deallocated since the write depends on all of the reads and all subsequent references depend on the write.

The trace pile contains the nodes of the execution tree in the same order as in the instruction trace. For each node `n`, `n.line` is the source line associated with the instruction (leaf) or procedure call (internal) corresponding to `n`, `n.parent` is the parent node in the execution tree and, if `n` is on the path between the root node of the tree and the most recent event (leaf node), `n.next` is the last child of `n` (the node one step closer to the most recent one along that path), so `n.next.parent` = `n`. This path corresponds to the call stack; `n.next` is the stack frame on top of `n`.

For each instruction level dependence found using the memory table, a source level dependence is computed using

Prog	#L	#sD	No RLC #iD	RSz	T	RLC #iD	RSz	T
ex	17	15	3.7K	12K	0.8	3.7K	12K	0.8
fib	22	7	32M	14K	6.7	32M	14K	6.9
qs	79	83	82M	39M	24.7	82M	35M	22.7
mpeg	6053	3330	3.3G	2.0G	2355	3.2G	109M	847

RLC: Read List Compaction, #L: #non blank source lines, #sD: #source dependences, #iD: #instruction leve dependences, RSz: max bytes for read lists, T: run time (s)

Table 1. Some experimental results

the algorithm in Figure 4. Here we have made use of the fact that the NCA must be part of the path from the late instruction level endpoint to the root node in the execution tree. Thus we can search from the early endpoint towards the root; the first node on the stack is the NCA.

Path compression can be used to decrease the number of iterations of the `while` loop. Every node `n` that is visited but is not on the stack can have its `parent` field set to its closest ancestor on the stack. We conjecture that this reduces the complexity of the algorithm to essentially constant time.

We can do better than path compression by *compacting* the trace pile. Once a procedure call has returned, we will not distinguish between different events in the subtree corresponding to its (completed) execution. They will all be represented by the root node of the subtree. We periodically compact the trace pile, replacing subtrees corresponding to completed calls by their root nodes. Since the memory table contains pointers into the trace pile, compaction entails forwarding these pointers to the root nodes of the compacted subtrees. After compaction, the trace pile contains the tree nodes corresponding to the stack and their immediate children, with non-leaf subtrees abridged to just the call and return events.

After forwarding, pointers to previously distinct events in the same read list now may point at the same tree nodes. In this case it is unnecessary to represent more than one copy of each pointer, thus compacting the read lists. This optimization is crucial in practice.

3.3 Prototype implementation

Embla is based on instrumented execution of binary code. Although our examples of profiling output use a high level language (C), the profiling itself is on the instruction level, followed by mapping the information to the source level using debugging information in the standard way.

Embla uses the Valgrind instrumentation infrastructure, so there is no offline code rewriting; the Embla tool behaves like an emulator of the hardware.

4 Preliminary Experiments

We have run some preliminary experiments to verify that the tool performs as expected. Some results are reported in Table 1. The programs are `ex`, our example from Figure 2, `fib`, a recursive Fibonnaci implementation, `qs`, a recursive quicksort implementation and `mpeg`, an MPeg encoder [11] encoding 10 frames.

It is interesting to see that read list compaction has a very different effect on different programs depending on the frequency of long sequences of reads from the same location (the RSz columns), ranging from no difference for `ex` and `fib` to a 20x difference for `mpeg`. We also see that the number of instruction level dependences (the #iD columns) are affected since the read lists are shorter when a write occurs following compaction (however, the eliminated read list items would have yielded no new source level dependences).

We note that Embla finds the expected independence of the recursive calls in the recursive divide-and-conquer programs `qs` and `fib` although `qs` is an in-place version coded with pointers and pointer arithmetic, something that is well known to be difficult to deal with for static analyzers. We have also done naive hand parallelization of these programs according to the information yielded by Embla and obtained the expected good speedup.

4.1 Analysis of dependencies in GCC

We are ultimately interested in knowing how well the dependencies collected during testing predict the dependencies encountered in production runs. Since the sequential program we test is deterministic, the issue becomes how well the behaviour under the testing inputs predict the behaviour under other inputs.

To shed some light on this issue, we have studied the SPEC CPU 2006 benchmark 403.gcc, a variant of the popular open source Gnu C Compiler. This is a very challenging program since, being an optimizing compiler, it essentially looks for a large set of patterns in its input an applies different code to different patterns.

We have collected flow (RAW) data dependencies for 88 different files (6–6328 lines of pre-processed C code) drawn from the code base of 403.gcc itself. These dependencies do not include dependencies due to `malloc` and friends.

Each dependence is a triple $(f, l_1 \leftarrow l_2)$ meaning that l_1 in source file f depends on line l_2 in the same file (recall that a dependence is always a pair of lines in the same function). We only look at forward dependencies where $l_1 > l_2$. Backward dependencies occur only in loops and the current Embla prototype does not handle loop-carried dependencies adequately. In particular, Embla does not recognize induction variables, hence reporting all loops as having loop-

Filter	# Dependencies					Size of deltas					Relative sizes (%)			# ∅
	max	min	avg	total	% of N	max	min	avg	total	% of N	max	min	g-mean	
All 403.gcc source files														
N	57733	8070	37589	80876		492	0	74	6502		1.08	0.00	0.09	9
T	39654	5637	26654	58336	72.1	434	0	41	3630	55.8	1.33	0.00	0.06	14
`c-parse.c`, `combine.c`, `insn-recog.c`, and `toplev.c`														
N	13218	718	7099	22158		215	0	33	2885		1.92	0.00	0.24	15
T	6499	636	3657	12001	54.2	134	0	13	1154	40.0	2.96	0.00	0.18	28
C	5008	398	2978	7166	32.3	64	0	6	568	19.7	1.53	0.00	0.14	40
CT	3139	347	1901	4936	22.3	39	0	4	331	11.5	1.44	0.00	0.22	56

Filters are **N**one, **T**ransitive or **C**ontrol flow, # Dependencies and Size of deltas give maximum, minimum and average sizes of D_i and D_i^*, respectively, over $i \in \mathcal{A}$ as well as sizes of $\cup_{i\in\mathcal{A}} D_i$ and $\cup_{i\in\mathcal{A}} D_i^*$. % of N gives the total for the row as a percentage of the total of the N row. Relative sizes give the maximum, minimum and geometric mean size of D_i^* as a percentage of D_i. Finally, # ∅ gives the number of inputs that yielded empty D_i^*.

Table 2. Dependence statistics

carried dependencies. Loop bodies may still be parallelized using only forward dependencies.

The 88 inputs yield a total of 80876 dependencies, which can be compared to the 484930 C source lines (excluding headers) of this build of GCC. Evidently, the dependencies are rather sparse due in part to the large number of lines not containing memory references at all, and the number of unique source lines that occur in some dependence is 49196 (hence lines that occur in dependencies do so in on the average about 1.6 lines, another measure of their sparseness).

Since we do not know the set of all dependencies that could be provoked by some input, we give dependency deltas, sets of dependencies that are generated by some input but not with some reference set of inputs, rather than presenting coverages as percentages of all dependencies. Thus, if the dependencies found with input data set i is D_i, we study

$$D_j^I = D_j \setminus \bigcup_{i\in I} D_i$$

for some set of inputs I not containing j. In particular, we have $D_j^* = D_j^{\mathcal{A}\setminus j}$ where $\mathcal{A}$ is the set of all the 88 inputs.

Table 2 shows some statistics on the dependencies found in 403.gcc. From the relative sizes columns we see that at most about 1% of the dependencies in any D_i are preserved in D_i^*; these are the dependencies that are only provoked by input i. The # ∅ column shows that 9 out of the 88 D_i^* were empty, meaning that these inputs generated no dependencies that were not generated by other inputs.

Looking at the dependencies in the D_i^* sets, we noticed that many of them were transitively implied by others. That is, for many $(f, l_1 \leftarrow l_2) \in D_i^*$ there were $(f, l_1 \leftarrow l), (f, l \leftarrow l_2) \in D_i$ for some l. These dependencies give the same constraints on parallelization and make $(f, l_1 \leftarrow l_2)$ redundant. The T filter removes these redundant dependencies, yielding a 28% reduction in overall dependence numbers and a 44% reduction in the D_i^* sets.

We next noticed many data dependencies that were subsumed by control dependencies (which Embla currently does not capture). To get an idea of the magnitude of this contribution, we implemented an approximate control flow analysis by hand for four modules of 403.gcc that were top contributors to the D_i^* sets and appeared to have nontrivial control flow. We also moved dependence endpoints up in the syntax tree, e.g. from a branch in an if statement to the if statement itself in a fashion analogous to the NCA computation discussed in section 3.2.

The results are given in the four bottom rows of Table 2 where we focus exclusively on dependencies $(f, l_1 \leftarrow l_2)$ where f is one of the four modules. We see from the first row that these modules account for almost half of the size of the D_i^* sets. Looking at the rightmos column, we see that control flow filtering alone makes almost half of the D_i^* sets empty, and that together with transitive filtering almost two thirds are eliminated.

To account for the remaining D_i^* sets we turned to coverage analysis using the `gcov` tool. We form L_i and L_i^* sets containing lines executed in the same way as for dependencies above. Figure 5 plots the size of L_i^* along the x-axis and D_i^* along the y-axis. The left hand plot shows the unfiltered data for all modules wheras the right hand one shows the data for the CT-filtered four modules. The left hand plot shows a very clear correlation between execution coverage and dependence coverage wheras the right hand plot is less conclusive; it will be interesting to see the effect of control flow filtering of all modules. However, of the 30 inputs that yielded empty L_i^* sets, only one had a nonempty (actually singleton) D_i^* set.

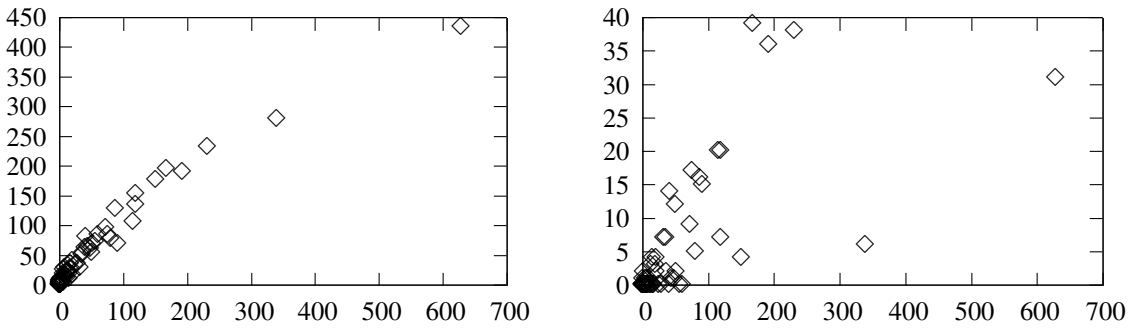

Figure 5. Coverage deltas (x-axis) versus dependence deltas (y-axis) for no (left) or CT filtering

5 Conclusion

We have presented Embla, a data dependence profiling tool, discussed the main issues and the algorithms used to resolve them. We argue that data dependence profiling is a useful and practical complement to static analysis for parallelization of new and legacy code. Finally, we have reported on a few initial experiments with Embla showing a strong correlation between executing the same part of a program and generating the same dependencies.

In the future we plan extensions and refinements of Embla as well as the application to real legacy code and more parallelization experiments using parallel environments such as OpenMP.

References

[1] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Comuting*, 37(1):55–69, August 1996.

[2] M. Cintra and D. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 38(10):13–24, 2003.

[3] M. Conway. A multiprocessor system design. In *Proceedings of AFIPS FJCC'63*, volume 24, pages 139–148. Spartan Books, 1963.

[4] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[5] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: a Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[6] D. Lea. A Java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, New York, 2000. ACM Press.

[7] D.K. Lowenthal and V.W. Freeh. Architecture-independent parallelism for both shared- and distributed-memory machines using the Filaments package. *Parallel Computing*, August 2000.

[8] N. Kim *et al.* Leakage current: Moore's Law meets static power. *IEEE Computer*, 36(12):68–75, December 2003.

[9] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the* 7^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11. ACM Press, 1996.

[10] M. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the* 9^{th} *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2003.

[11] MPEG software simulation group. Reference MPEG-2 video codec software. http://www.mpeg.org/MPEG/MSSG/.

[12] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *The* 22^{th} *Annual Int. Symp. on Computer Architecture*, pages 392–403, 1995.

[13] N. Vachharajani, M. Iyer, C. Ashok, M. Vachharajani, D. August, and D. Connors. Chip multi-processor scalability for single-threaded applications. *SIGARCH Computer Architecture News*, 33(4):44–53, 2005.