Cit: A GCC Plugin for the Analysis and Characterization of Data Dependencies in Parallel Programs

Sumeet S. Kumar, Anupam Chahar, Rene van Leuken
Circuits and Systems Group, Faculty of EEMCS,
Delft University of Technology,
s.s.kumar@tudelft.nl

***Abstract*—Management of shared data accesses by concurrent tasks is a challenging aspect of parallel programming. In this paper, we present the *Cit*, a plugin for GCC that performs compile-time analysis and characterization of data dependencies between concurrent tasks within parallel programs. Dependencies are classified as *Always Conflict (AC)* and *May Conflict (MC)* based on the likelihood of them resulting in a conflict, according to the nature of their reference, their location within the control flow and their dependence on input data. Furthermore, Cit reports load-store volumes for concurrent tasks, thus enabling estimation of their approximate data set size. The Cit plugin is shown to accurately detect data dependencies within standard benchmarking workloads and characterize all conflicts that could arise due to them. The entire analysis is performed during compilation, and incurs a runtime ranging from a few seconds to a few minutes. The results of Cit's analysis form a critical feedback to programmers, allowing visualization of program data dependencies, and enabling exploration of inherent parallelism.**

I. Introduction

Parallel programming in the many-core era is an inherently complex task requiring careful management of shared data accesses. Applications that derive large benefits from high performance many-cores often exhibit significant amounts of *Disjoint Access Parallelism (DAP)* in their critical sections [1]. Non-disjoint concurrent accesses by concurrent tasks necessitate the enforcement of mutual exclusion using locks to prevent data races and incorrect execution. The pessimistic use of locks however leads to poor speedup despite the available computing power. The granularity and placement of shared data locks are thus critical towards exploiting the potential of modern many-core processors.

Transactional Memory (TM) [2] simplifies this by eliminating lock-based critical sections within concurrent tasks, instead replacing them with atomic transactions. Accesses to shared data otherwise protected by locks, are detected in TM systems at runtime through a conflict detection scheme, which in the event of a conflicting memory access, results in all but one amongst the contending transactions to abort and restart their work. Although the replacement of locks with such speculative transactions decreases programming complexity, it results in degraded execution performance in the case of large transactions that repeatedly perform conflicting memory accesses. For a pair of concurrent transactions that perform updates to different elements of a shared data structure based on the outcome of a conditional test on some input data, the target of each memory access made by the transactions is determined according to the branch. Consequently, while there is the possibility of the two transactions conflicting due to non-disjoint accesses, it is also possible that the two are often independent. Thus dependencies between transactions can be characterized as either *Always Conflict (AC)* or *May Conflict (MC)* depending on the likelihood of them actually resulting in a conflict during execution.

In this paper, we present the *Cit* plugin for GCC, that analyzes and characterizes dependencies between concurrent tasks in parallel programs at compile-time. Data dependencies between concurrent tasks are determined through a combination of custom intra- and inter-procedural analysis passes that analyze data use and control flow within the program. Dependencies are thus identified as type AC or MC based on the nature of their reference, presence of conditional branches, input data dependence, and their location within the control flow. Furthermore, Cit provides a visualization of the program structure and data sharing using generated graphs, and reports estimated load-store volumes for analyzed compilation units and the presence of common TM-specific performance pathologies in the program code. Cit is intended as an aid to programmers developing applications for parallel environments. The plugin provides critical feedback regarding the structure of the program as well as the nature and location of dependencies between its constituent tasks. As such, these results could be used in the parallelization of sequential code, the placement and refinement of shared data locks, analysis and optimization of C code for high-level hardware synthesis, and towards the selection of suitable conflict management strategies based on dependency types and workload characteristics in the case of TM applications. In this paper, we focus primarily on the characterization of TM programs.

The rest of this paper is organized as follows: Section II provides an overview of the state of the art. Section III introduces the Cit plugin and describes its constituent analysis stages and Section IV its implementation in GCC. Section V experimentally validates the plugin, and reports the results from its analysis of transactional workloads including those from the STAMP benchmark suite, alongside their generated data dependency graphs. Finally, Section VI provides concluding remarks on the presented Cit plugin.

II. Related Work

A number of techniques exist in literature for the analysis of dependencies in parallel programs. A significant portion of such literature is devoted to approaches that use execution profiling to characterize applications and detect data dependencies. Embla [3] is one such tool that uses execution profiling to detect data dependencies in sequential programs, and thus indicate opportunities for parallelization. QUAD [4] on the other hand analyzes memory accesses by communicating functions to determine potential data dependencies. Such approaches however require a wide variety of data sets and multiple execution runs in order for their results to represent common-case application behaviour. Static analysis techniques on the other hand provide a worst-case estimate of an application's dependencies, with a shorter runtime. SvS [5] is a technique that uses data dependency information gathered through static analysis in determining an optimal task schedule for parallel programs. The technique essentially determines the set of possible data dependencies that can occur in the program through a compile-time reachability analysis, and later refines this set through execution profiling at runtime. The *approximate* nature of the static analysis does not take the task's control flow into account and thus yields very conservative dependency estimates. Increased accuracy at this stage may improve execution performance by decreasing the incurred overheads due to runtime refinement. Mannarswamy and Govindarajan presented schemes to improve the performance of software transactional memory through compiler-assisted lock assignment [6], and through the compiler-directed handling of certain conflicts between transactions [1]. Both these schemes rely on the standard inter-procedural analysis of the Open64 compiler, and its generated data dependency graph. Transactions are classified based on this analysis as either *always conflicting* or as *disjoint*. The existence of *may conflict* dependencies causing transactions to *sometimes* conflict however, is not considered in this analysis, resulting in pessimistic dependency estimates. Cit on the contrary, categorizes dependencies according to their location within the control flow and takes into account the presence of conditional branches and input dependence while estimating the likelihood of dependencies causing a conflict.

III. Cit Architecture

Understanding the data usage patterns of concurrent tasks is essential in determining possible data dependencies within the program. We consider concurrent tasks to be transactional functions containing assignment statements as well as function calls, and uncovering data dependencies between them requires an analysis of each function as a separate unit. Cit uses intra-procedural analysis to gather data definitions such as assignment operations and variable usage, as well as control flow within functions. This yields a detailed snapshot of each function within the program and its variables. Inter-procedural analysis uses such snapshots in addition to the list of calls from the call graph in its analysis, and thus determines possible data dependencies between functions and their type (AC/MC). The worst-case load-store volume for each function, and detection of potential performance pathologies are also obtained from the captured data and control flow for each function. Figure 1 illustrates the four analysis stages of the Cit plugin.

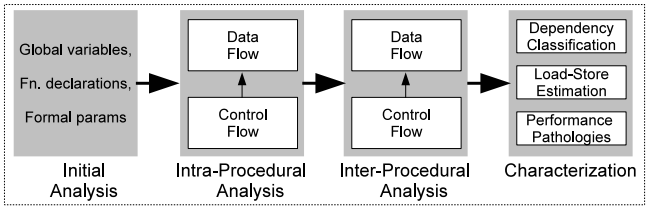


Fig. 1. Stages of the Cit analysis methodology

A. Initial Analysis

In the first stage, a basic analysis of the application is performed in order to gather global variable declarations, and formal parameters of functions. These are necessary during inter-procedural data flow analysis for identifying dependencies that result from calls to specific functions. Both, the formal parameters corresponding to functions as well as global variables are logged in a database for use in subsequent stages.

B. Intra-Procedural Analysis

During this stage, each function in the application is analyzed individually in order to determine local control and data flow, load-store volume and the presence of branches. A set of *define-use/use-define* relationships are built through the analysis of each function body. Since such liveness analyses consider execution of the function body to be sequential, all definitions for a variable preceeding the newest are killed. However, data dependencies always remain, irrespective of the liveness of the definition they result from at any given point in the function's execution. This is especially the case when a new definition uses an older definition that it subsequently replaces. Killed definitions are therefore also taken into account while building the dependency graph during inter-procedural data flow analysis. Indirect references using pointers form a complex part of this analysis since their actual targets cannot be accurately determined at compile-time unless such references use constant offsets. In such cases, ignoring dynamic offsets altogether still yields a basic dependency relation with respect to the base of the data structure being referenced. Inter-procedural dependencies may exist in the form of arguments in function calls and pointer references within the body of the function being analyzed, or in global variables. Cit therefore extracts the call graph of each analyzed function together with the arguments passed to each callee within the function body, and maps these to the formal parameters previously logged in the database during Initial Analysis. This ensures that dependencies can be tracked deep into callee functions, from where they could potentially cause global non-disjoint accesses. In addition to dependencies, Cit also logs the size of load-store operations that occur within functions. The control flow when overlaid with this information yields estimates of load-store volumes along each control path in the function. The estimation however is in terms of basic blocks alone, and therefore, in the event of an intermediate function call, the corresponding worst-case load-store path in the callee function is included in computing the calling function's load-store volume.

C. Inter-Procedural Analysis

The previous stage analyzed individual functions to determine internal data flow, variable use and all constituent

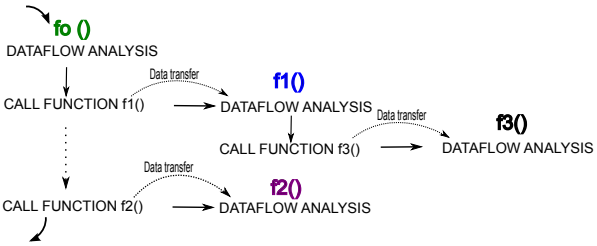


Fig. 2. Overview of Inter-Procedural Analysis

function calls. During inter-procedural analysis, the interplay between these program elements is determined through the exploration of the context sensitive call graphs starting from the topmost entry point, resulting in a deep call graph for each concurrent task composed of the various functions that it calls during its lifetime. The data flow and definitions within each function of this call graph are analyzed to determine whether operations as well as the function's arguments reference local or remote data. In the latter case, each variable is iteratively tracked to its source, i.e. a shared global datum with potentially multiple remote definitions, and logged in the function's variable database. The analysis continues into each constituent function within the call graph after the entry point as illustrated in Figure 2. Note that for every call encountered after the entry point, the calling context from the caller is retrieved and transferred to the data flow analysis of the callee function. The variable database obtained for each constituent function *f1()*, *f2()* and *f3()* in Figure 2 is aggregated and thus considered to be part of *f0()* since concurrent instances of *f0()* may become inter-dependent through these functions. This property of Cit can also be used to evaluate *thread safety* of functions in parallel environments.

Dependencies between concurrent tasks are finally established by a comparison of their aggregate variable databases. Therefore, the set of dependencies D between any two tasks Ti and Tj is:

$$D = D_{Ti} \cap D_{Tj} \tag{1}$$

where D_{Ti} and D_{Tj} represent the aggregate variable databases of concurrent tasks Ti and Tj respectively. Note that $D = \emptyset$ for non-disjoint reads.

D. Characterization

A novel feature of Cit is its characterization of data dependencies according to the program control flow as AC or MC. In this final stage, the program is characterized to determine the nature of data dependencies, and estimate the volume of load-stores within each concurrent task.

1) Dependency Characterization: At the end of inter-procedural analysis, the dependency relationships between functions as well as the internal control flow within each are known. Together, the two enable characterization of inter-procedural dependencies according to the likelihood of their occurring at runtime and thus causing a conflict. Dependencies are categorized based upon their location within the control flow, and the nature of the reference that they result from. Consequently, they may be of two types: *Always Conflict (AC)* and *May Conflict (MC)*.

The first type results from data dependencies that arise due to static references, i.e. direct or indirect with constant/input independent offsets. Concurrent execution of multiple instances of a function containing such a dependency will always conflict in their accesses to the contentious shared datum. On the other hand, the presence of conditional branches in the control path may result in the shared access not occurring at all. In this case, the dependency is categorized as MC. The latter type is also inferred for dependencies within loops with input dependent bounds, as well as for statements with input dependent operations. Indirect references that utilize constant constant offsets are categorized as AC in absence of the above mentioned conditional branch or loop conditions, while those with dynamic or input dependent offsets are categorized as type MC and are referred to as dynamic dependencies.

Thus dependencies are of type:

- *AC* for static references without any input dependence, or conditional branches and input dependent loops
- *MC* for all references preceeded by a conditional branch, or within a loop with input-dependent bounds, and for indirect references with dynamic offsets

2) Load-Store Estimation: Lazy versioned TM systems typically isolate speculative writes until they are determined as disjoint, or as legal writes in the case of contention. It is thus critical for the volume of such writes to remain within the capacity of the write-buffer in order to avoid performance degrading overflows. Data set size estimates at compile-time enable such an evaluation, and in conjunction with dependency information, provide a suitable basis for program optimizations to improve execution performance. Load-store volumes for each concurrent task are computed in the same manner as during Intra-Procedural Analysis, using its internal control flow, call graph, and individual volumes for each callee function. Loops with input dependent bounds however present a source of inaccuracy to Cit's load-store estimates, in which case the data set size is computed considering only a single iteration of the corresponding loop section. We expect to overcome this limitation in subsequent versions of the plugin.

3) Performance Pathologies: Apart from write-buffer overflows, a number of other performance pathologies and potential pitfalls may exist within the program, often without any perceivable effect in the common case. However, these may form significant performance bottlenecks under specific circumstances.

- *Dependencies within loops:* TM systems using optimistic conflict detection delay the detection of data dependencies until the end of the transaction's execution. In the case of transactions that execute long running loops with dependencies within the first few iterations, this policy causes execution to continue needlessly. Such dependencies may therefore present a performance bottleneck, and result in wasted work.
- *Recursive functions:* Recursive functions that contain inter-procedural dependencies may result in infinite call sequences if incorrectly used. Furthermore, such sequences can potentially cause write-buffer and stack overflows in the system. From a debugging perspective, this information is useful in locating instances of unplanned recursion.
- *Memory management functions:* Dynamic memory allocation within parallel tasks with a monolithic heap

allocator can also act as a performance bottleneck, with each task waiting on the heap lock.

The presence of these pathologies is determined using the control flow and inter-procedural data dependency graphs, and is reported as part of Cit's characterization reports.

IV. Implementation in GCC

Cit is implemented as a plugin for *GCC-4.5* [7], allowing code under compilation to be analyzed and manipulated without modifying the compiler. During compilation, high-level program code is converted into the *GIMPLE* internal representation which is used by GCC's optimization passes [8]. This internal representation is a simplified form of the program code, with complex expressions reduced into a sequences of GIMPLE tuples that indicate the expression type, operations and the appropriate operands. Function calling contexts, variable declarations and program constructs such as branches are trivial to obtain from the GIMPLE representation, and their analysis is simplified by a wide range of internal macros within GCC. Although the entire analysis is performed at the granularity of basic blocks, the code section that is primarily targeted for dependency analysis may be selected by specifying the unit's name as an argument. At the start of the GIMPLE pass, the complete internal representation of the compilation unit is dumped by Cit into intermediate files for analysis. GIMPLE statements are processed using the *FOR_EACH_BB* iterators, with *function at a time* granularity. A subsequent run through the statements yields an inter-procedural call graph, allowing the control flows from the first analysis to be expanded in place. Finally, data flow analysis is used to determine data dependencies between concurrent tasks as previously explained.

An overview of the compilation process highlighting the placement of Cit is shown in Figure 3. Cit is instantiated between GCC's *Visibility* and *Static Single Assignment (SSA)* optimization passes. At the end of the visibility pass, the application code is trimmed of unreachable functions and statements. Further, at that point in the compilation, the representation has far fewer temporaries than the SSA form, thus reducing both the runtime as well as the complexity of the dependency analysis without reducing Cit's coverage. Since inter-procedural optimization are typically performed only after the GIMPLE pass, the placement of the plugin at this point presents an opportunity for optimizations to be driven by the results of Cit's analysis.

The plugin produces text files with characterization reports for the analyzed code, listing dependencies between functions together with the line numbers in the source code at which they exist, allowing programmers to verify their code with relative ease. A DOT graph description language representation of control flow, call and data dependency graphs is also produced alongside the plain text reports. These may be used in conjunction to identify independent sections of code that contain exploitable parallelism. The runtime of the Cit plugin ranges from a few seconds to a few minutes depending on the size of the code under compilation. A limitation of our implementation is that the analysis requires application code to be included into a single file, allowing its analysis as a single compilation unit. However, conversion of a multiple source

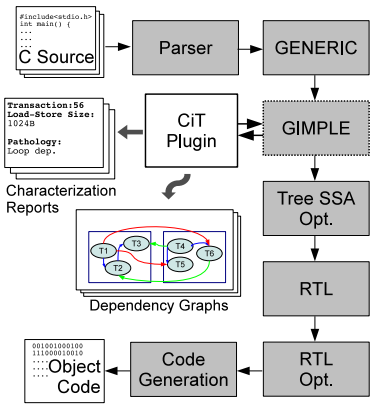


Fig. 3. Compilation flow illustrating the placement of Cit's analysis passes

program into such a single unit is simple to perform using the *Merge Program* feature of the CIL framework [9].

V. Experimental Evaluation

A custom transactional test application incorporating six micro-kernels was used to validate Cit, since its dependency graph and characteristics were both known *a priori*. The micro-kernels included data dependencies in the form of global variables, function parameters passed by reference, pointers to dynamically allocated memory. The application's concurrent tasks 1 through 4 are loop-based code sections that perform read-modify operations on individual variables, array elements and structures that are potentially shared, from within transactions. Tasks 5 and 6 represent node generation and key modification functions for a linked list. The *a priori* data dependency graph for the test application is presented in Figure 4.

The application was compiled with the Cit plugin enabled, and the obtained data dependency graph with respect to *task 1* is shown in Figure 5. This graph is observed to match the a priori dependency graph in both location as well as type of data dependencies. However, the graph indicates an additional dependency in the program due to concurrent accesses to an array by tasks 1 and 4. These accesses are in fact disjoint, but since they are performed as indirect references using the loop variable, their address offsets are ignored, thus yielding a basic dependency relationship considering the base of the array alone. However, this dependency is marked as *MC* since it may not actually result in a conflict if the concerned references are disjoint, i.e. the loop variables have different initialization values. The first column in Table I lists the characterization report for the test application. The dynamic dependencies as a result of indirect references by tasks 5 and 6 are marked as MC in the report. Included as meta-data, this information could

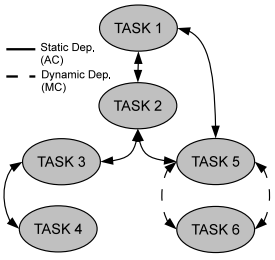


Fig. 4. Data dependencies within the custom *test* application

TABLE I. CHARACTERIZATION REPORTS FOR ANALYZED WORKLOADS

	TEST+	KMEANS	BAYES		NB-LL
Analyzed concurrent task	***all***	***work***	***learnStructure***	***createTasklist***	***user***
Static Dependencies	6	2	5	2	0
- Always Conflicting (AC)	5	0	1	0	0
- Within loop (MC)	1	2	5	2	0
- Within conditional branch (MC)	0	2	4	2	0
Dynamic Dependencies (MC)	2	1	0	0	2
Worst-case load-store count	32	78	2870	1122	140
Recursion	No	No	Yes	No	No
Mem. Management Functions	Yes	No	Yes	No	Yes

+This characterization report aggregates all concurrent tasks in the application.

aid in pinpointing tasks whose transactions require validation, and those that can benefit from simple data forwarding. The load-store estimates for the application considering a single-iteration of all its constituent loops, was found to be accurate from the GCC output assembly listing.

The plugin was subsequently used to analyze the *kmeans* and *bayes* workloads from the STAMP benchmark suite [10], and the results were verified against those from literature [1]. In addition, a non-blocking linked list implementation [11] was also analyzed. These analyses explored concurrent tasks within each workload, tracing their dependency types and relationships, as well as their load-store characteristics and performance pathologies. The characterization reports for these applications are also listed in Table I. The *kmeans* workload groups objects in an N-dimensional space into K clusters. Updates to cluster centers are performed iteratively, and are protected by a transaction. Parallelized execution of *kmeans* consists of multiple threads concurrently executing the *work* function that performs this operation. The structure containing cluster centers is shared between these threads. Mannarswamy et al. in their analysis of *kmeans* [1], resolved the update operation as an AC dependency, inferring that accesses to the shared data structure are always non-disjoint. However, Cit's characterization reports indicate this dependency to be of the type MC. This is due to:

1) The location of the dependency within a conditional branch nested inside an input dependent loop section. This may influence the range of cluster centers modified by the operation, and thus the occurrence of conflicts between the concurrent threads.
2) The dependence of the cluster selection on the arguments to the *work* function. Therefore, based on the input, concurrently executing threads may update disjoint cluster centers. The dependency, consequently, is dynamic.

Describing the dependency as always conflicting is pessimistic, since the likelihood of two threads simultaneously attempting to update the same cluster center is low. This observation is further supported by [10], thus indicating that conflicts between *kmeans* threads are not guaranteed to always conflict. There also exist static MC dependencies that involve direct references to shared variables *global_i* and *global_delta*, protected by small read-modify-write transactions. Stores to *global_i* are seen to be preceeded by a conditional branch in Figure 6, indicating that potential conflicts are predicated on the outcome of that branch. In this case however, the impact of such conflicts can be expected to be low considering the size of the transactions involved. An overview of the analyzed *kmeans* workload is shown in Figure 7, with both call and data-dependency information. Functions such as *work* executed concurrently by multiple threads may conflict on account of potentially non-disjoint accesses. This is indicated by means of a dependency relationship originating and terminating at the same function. The *bayes* workload on the other hand exhibits more complex characteristics. It implements a learning algorithm that detects dependencies between variables in a *Bayesian network*, progressively expanding it by adding sub-graphs of dependent variables. The likelihood of each dependency is recursively computed and tracked by an *ADTree* structure. *Bayes* uses multiple threads of execution to speed up the learning algorithm. Each thread computes dependencies for variables, and attempts to update the shared network structure through a transactional operation. There are two primary

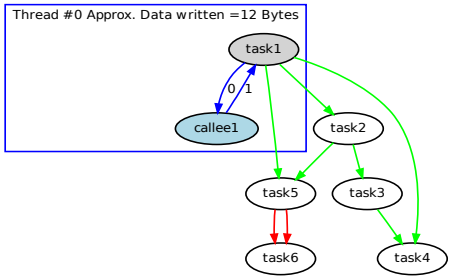
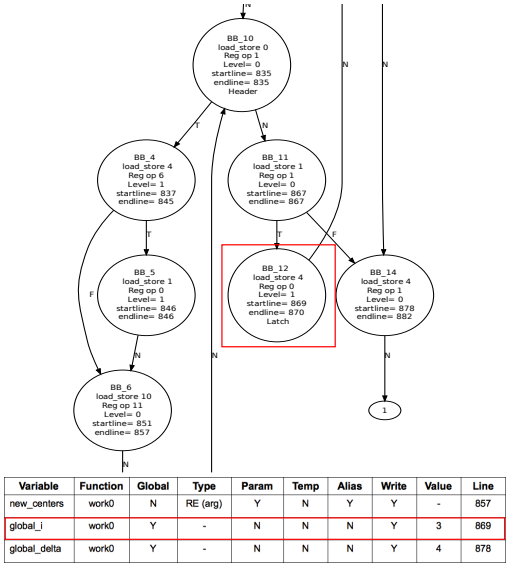


Fig. 5. Call graph for *task1* overlayed with data dependency information for the *test* application



Variable	Function	Global	Type	Param	Temp	Alias	Write	Value	Line
new_centers	work0	N	RE (arg)	Y	N	Y	Y	-	857
global_i	work0	Y	-	N	N	N	Y	3	869
global_delta	work0	Y	-	N	N	N	Y	4	878

Fig. 6. Control flow graph of the *kmeans work* function highlighting a shared write under a conditional branch. An extract of the variable database is shown in the table below the control flow graph.

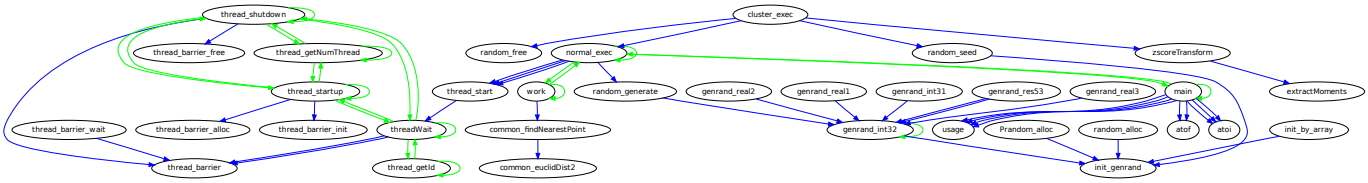


Fig. 7. Call-graph overlayed with data dependencies for the *kmeans* workload. The *work* function here is a concurrent task, and bears dependencies only with it's parent (*normal_exec*) and its own concurrent instances.

threads within *bayes*: *createTasklist* and *learnStructure*. The former uses transactions to read and update the likelihood of a dependency for each variable, and subsequently determine the next task to analyze. The *learnStructure* thread in turn reads from the shared task list, and determines the operation to perform on the network. This is implemented in the code as a switch statement with cases determining whether edges are to be inserted or removed from the network.

Cit's analysis reports reflect the effect of this operation on *learnStructure*'s dependencies, which are consequently classified as type MC. In the case of concurrent execution of multiple threads of this function, each may attempt to perform a different operation on the same node of the network, and thus cause a conflict. *learnStructure* is also observed to be dependent on *createTasklist* through the *taskPtr* data structure that is populated by the latter. In general, the presence of a large number of dependencies (as compared to *kmeans*), as well as the interdependence of threads of both functions are indicators of the high contention inherent in the *bayes* workload.

Load-store estimates in the characterization reports illustrate the data-set size for threads in each workload. *kmeans* for instance is observed to have a significantly smaller data-set per iteration than both *bayes* threads. The number and nature of loops and branches present within each thread further provide an indication of runtime. In the case of *learnStructure*, Cit reported an unconditional loop containing transactions that perform operations on the network, thus indicating a potentially long runtime. The characterization reports therefore reflect the following design characteristics of the *kmeans* and *bayes* workloads [10]:

- *kmeans*: Small data-set, low contention, short runtime (no recursion)
- *bayes*: Large data-set, high contention, long runtime (recursion)

An non-blocking linked list benchmark (*NB-LL*) was also analyzed using the Cit plugin. The *user* function within this workload performs insertion and deletion operations on a single list using memory management functions. The two dynamic dependencies associated with the indirect references to the linked list during insertion and deletion are correctly identified by Cit's analysis passes. The characterization report for NB-LL is listed in Table I.

VI. Conclusion

In this paper, we presented Cit, a GCC plugin for compile-time analysis and characterization of data dependencies within parallel programs. In addition to accurately determining data dependencies within standard benchmark applications, Cit also correctly classified these as either Always Conflict (AC) or May Conflict (MC) based on the likelihood of them resulting in a conflicting memory access. This was highlighted in the case of the *kmeans* workload from the STAMP benchmark suite, in which a critical dependency known from literature to be of type AC, was correctly identified by Cit as type MC instead. This observation was supported by the application's known design characteristics, as well as its execution behavior. Cit's features make it a powerful tool for fast analysis and visualization of parallel programs, providing programmers with the critical feedback they need to optimize code, improve execution performance and explore potential parallelism in their applications.

VII. Acknowledgements

This work was supported in part by the CATRENE programme under the Computing Fabric for High Performance Applications (COBRA) project (CA104).

References

[1] S. Mannarswamy and R. Govindarajan, "Handling conflicts with compiler's help in software transactional memory systems," in *Proceedings of the 39th International Conference on Parallel Processing*, 2010, pp. 482 –491.

[2] M. Herlihy and J. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th annual International Symposium on Computer Architecture*, 1993, pp. 289–300.

[3] K.-F. Faxen et al., "Embla - data dependence profiling for parallel programming," in *Proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems*, 2008, pp. 780–785.

[4] S. A. Ostadzadeh et al., "Quad: a memory access pattern analyser," in *Proceedings of the 6th international conference on Reconfigurable Computing: architectures, Tools and Applications*, 2010, pp. 269–281.

[5] M. J. Best et al., "Synchronization via scheduling: techniques for efficiently managing shared state," in *Proceedings of the Conference on Programming Language Design and Implementation*, 2011, pp. 640–652.

[6] S. Mannarswamy, D. R. Chakrabarti, K. Rajan, and S. Saraswati, "Compiler aided selective lock assignment for improving the performance of software transactional memory," in *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, 2010, pp. 37–46.

[7] "Gnu compiler collection (gcc) 4.5." [Online]. Available: http://gcc.gnu.org/gcc-4.5/

[8] J. Merrill, "Generic and gimple: A new tree representation for entire functions," in *Proceedings of the GCC Developers Summit*, 2003, pp. 171–180.

[9] G. C. Necula et al., "Cil: Intermediate language and tools for analysis and transformation of c programs," in *Proceedings of the International Conference on Compiler Construction*, 2002, pp. 213–228.

[10] C. Cao Minh et al., "Stamp: Stanford transactional applications for multi-processing," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2008, pp. 35 –46.

[11] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, 2001, pp. 300–314.