A Data Dependence Graph in Polaris

Yunheung Paek Paul Petersen

July 17, 1996
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract

Data dependence analysis has become one of the most important constituents of loop-level parallelizing compilers. The information gathered from the analysis is needed to determine both the potential concurrency of loop nests and the legality of loop transformations concerning loop parallelization. The analysis is based on data dependence tests such as the constant test and the GCD test. Once the analysis is complete, the dependence information for the input program is stored for later demand. Since this dependence information is stored in memory, recalculation of the information in following passes is unnecessary, reducing the whole parallelizing compiler execution time. However because this information often requires so much memory space, sometimes up to thousands of Mbytes without strict memory management, it can run out of heap storage, and cause another problem: how do we store the information efficiently? Besides memory space, the time to fetch the necessary information is also another criterion of concern because faster access generally requires more memory space. In this paper, we design an efficient method of dealing with data dependence information in Polaris in terms of memory space and access time. In this method, we use a compromise between the two criteria to get better performance.

1 Introduction

The main purpose of a loop-level parallelizing compiler is to find as many parallelizable loops as possible because, with more parallel loops, the compiler can perform more time and space efficient loop scheduling on the machine. In parallel loops, iterations can be assigned to any available processor without any constraint on the order of loop execution. The executions of these arbitrary orders do not alter the final output, which is identical to that of a sequential execution(if the discrepancy due to round-off error of the hardware is disregarded). This congruency can always be ensured by the attribute that there is no data flow crossing iterations in parallel loops, that is, any statement of one iteration never uses the value of a variable that was defined by a statement(both are not necessarily distinct) in the previous iteration.

We can identify parallelizable loops with the assistance of data dependence analysis. Each dependence test that is applied in the analysis first collects all the static information obtainable such as constant values, the number of loop iterations, the loop variants and so on, and calls its data dependence decision routine to determine if a given loop is parallelizable. Except for special restricted cases, most of these tests give only necessary conditions for dependence. Since the tests can not give sufficient conditions, the declared dependencies can not always be proved, that is, each test can prove the dependence of specific confined cases only. Because of the lack of an exact test to handle every case, we usually need to apply more than one dependence test. Some tests are much more powerful than others, but also take much more time to run. The constant, the single-subscript GCD and the single-subscript Banerjee test[?][?]

are known as the fastest tests, running in linear time. On the other hand, there are exponential time algorithms, such as the Omega test[?] and the range test[?]. They are potentially extremely expensive tests but they have been shown to run in moderate time range on the average because they mostly handle cases with simple and regular access patterns. These complex tests usually cover broader cases. There also exist several other dependence tests[?][?][?] that have different classes of time complexity. The prominent reason that all these tests fail is that there are unknown values at compile time. So, if all these static tests fail because of unknown values, dynamic dependence tests[?] sould be the last test we can resort to at the cost of run time overhead. It is still undecided whether or not these run time tests can offset their cost with significant gain from successful parallelization.

An optimal sequence of the selected dependence tests as not yet known, but by rules of thumb fast and simple tests generally should run before the complex ones. In Polaris[?], a parallelizing compiler currently being developed by our group, we apply static data dependence tests in the order of simple Delta test[?], constant test, GCD test, Banerjee test and range test. However, this order is subject to vary in the future.

During the data dependence analysis, two statements are chosen and are given as input to the dependence tests. After going though all the tests, the dependence between the statements is determined. Once the dependence is assumed or proved, more specific information about the dependence is computed, including the dependence direction and the dependence type. Before performing the dependence test on another pair of statements, we store the newly generated information for later demand, and thus after the analysis is completed, the entire dependence information for the input program is stored into the memory. The problem with storing the whole information is that this information often requires so much memory space, sometimes up to thousands of Mbytes without careful memory management, that it runs out of heap storage. In our system, we bit-pack data dependence information to maximize memory utilization. When the information is stored into memory, it is encrypted to a series of bits and packed in memory.

The problem with the bit-packed storage management is that the time to retrieve and to update the encrypted information takes too long. So far, we have not been concerned with the compilation time but only application running time. But, as loop transformation techniques grow in number and in complexity, the compile time becomes more important(We sometimes need several hours to finish compilation). In order to curtail this time delay, we take advantage of the 2-level hierarchy structure for storing dependence information : *active and inactive.* If some information was not recently accessed or if it is not being used actively, it is called inactive. Active information is any informtion that is not inactive. Inactive information is stored in bit-packed storage efficient memory blocks and active information is unpacked copy of the inactive information in the form of linked lists. Unlike the inactive structure, the active structure obviously uses much more space to allocate pointers but provides fast retrieval and update time. This structure is based on our observation that compared to the size of the entire dependence information, only a small portion of the information is likely to be retrieved frequently for a short period of time. So, our hierarchy structure is grounded on a concept similar to that of memory hierarchy: *locality.*

In Section **??**, we briefly take a look at the Polaris project. In Section **??**, the basic idea of data dependence and its abstract representation is illustrated, and its physical representation in the form of hierarchy structure is also presented. Finally, we discuss the user interface through which the user can access the data dependence information in Section **??**.

2 Polaris

Since most of current parallelizing compilers have so far focused on several specific patterns of codes to parallelize them, it is not surprising to know that they succeed on small kernels but often fail on large applications. Polaris is a new type of parallelizing compiler that will overcome these limitations. The basic idea here is that such a large application is to be automatically parallelized by current compilers with a few new techniques. The new techniques that have been included in Polaris are inlining, array privatization[?], symbolic dependence analysis, induction and reduction recognition and elimination, and run-time dependence analysis. The performance of Polaris has been tested on Perfect Benchmarks program suite[?] and quite promising so far.

Polaris has a powerful basic infrastructure, called Internal Representation[?], for manipulating Fortran77 codes on which all those new techniques are based. IR consists of several functional layers in which the lower layers provides the higher layers with a high-level abstract view of its functionality, hiding all the complex details of its operations. At the lowest layer, IR has the form of an abstract syntax tree that contains the necessary information for a given program, and going up to the higher layers, it adds more sophisticated and higher-level functionality to provide methods for program transformations. With the support of the object oriented language C++, IR provides structural flexibility and powerful data-abstraction mechanism so that the development time for new optimization techniques inside Polaris. In Polaris, the C++ class, **ProgramUnit**, contains the whole information for a subroutine or a function such as a statement list, a symbol table and so on. Each table or list inside the ProgramUnit class also expressed in the form of a C++ class object. For example, **StmtList** is for a statement list, **Statement** for a statement and **Expression** for an expression. A complex expression is recursively defined by its subexpressions and thus Its structure resembles a tree of the Expression class objects.

Since Polaris has succeeded in gaining the major performance achievement on the Perfect Benchmarks, we will move on to some remaining issues: the representativeness of our program suite and the performance on real machines. We are now working on expanding the set of benchmarks to cover broader range of the patterns of real programs. We also works on the postpasses of Polaris that generate from the Polaris output the real codes executable on real machines such as SGI power challenge, Convex C3 and so on. When we deal with real performance, the parallelization is not only issue anymore, but other optimization issues, the locality on parallel execution for instance, are also to be considered. Currently, Polaris output is suited to global shared memory machines.

3 Data Dependence Graph and DDgraph

3.1 Data Dependence Graph

If a statement `S1` uses a value of the variable defined by other statement `S2`, `S1` is data-dependent on `S2`. We can represent these dependence relations between statements graphically: a *data dependence graph*, where nodes of the graph usually stand for statements in the program and directed arcs stand for dependence relations. In the graph, we can also represent additional information belonging to the dependence relations, such as the types, and the directions or the distances[?][?]. Here is a simple example of data dependence graph in Figure **??**. The plain arc, the arc with a small segment of line on, and the arc with an small circle on represent flow, anti, and output dependence, respectively. More details about these notions and the use of data dependence graphs are discussed in [?].

3.2 DDgraph

The granularity of a data dependence graph may be varied. The nodes might represent either larger elements or smaller ones. In Figure **??**, the nodes correspond to statements, and each arc represents the dependence between them. But, this information is sometimes so coarse that we can not handle effectively the dependence relations in the program. In that figure, the source and the target of flow dependence between `S1` and `S2`, and those of the anti dependence between them are different expressions, but we can't tell the difference from the graph. In our scheme, a node stands for an expression. We believe this expression-level dependence graph is powerful and detail enough to provide most of dependence information that the users want to get for code optimization.

Now, a new problem has risen: *how can we afford the size of a dependence graph?* One assignment statement contains at least 2 expressions, and if the expressions consist of several subexpressions, the

```
S1:  X = Y + Z
S2:  X = X / Z
S3:  Y = Z * 4
```

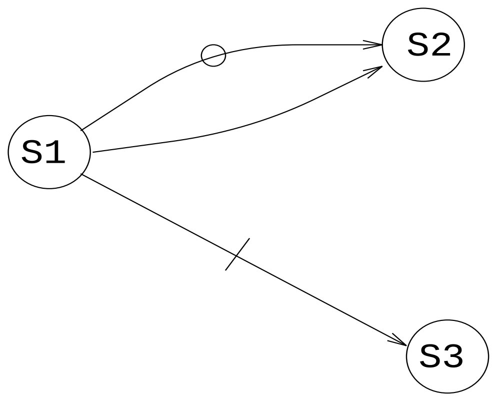

Figure 1: Data Dependence Graph

total number of expressions in a statement might reach up to more than 10. In the last figure, S1 has 4 expressions, that is, X, Y+2 and its subexpressions Y and Z. The characteristics of recursiveness of expressions enable a data dependence graph to become bigger. There are also several loop transformation techniques that makes the graph denser and bigger. For example, loop normalization[?] mostly transforms subscript functions more complex, and inlining[?] increases the size of a program primarily by adding to that of the loops. So, if we just simply assign a pointer variable to each arc and assign a character or an integer to each field of information, the dependence information for only one statement will even require more than 100 bytes on average. The straightforward way to reduce the memory requirement of a dependence graph is to tighten up the graph into a series of bits. But the reckless condensing raises other problems; long latency time to access and the functional expandability. Under this condensed structure, it takes too long to insert, delete and even retrieve some part of the graph. And it is also hard to implement new high-level functions or operations upon the structure. So, while still keeping the graph size small, we need a method which supports a fast access to the graph and the future expandability of graph access operations.

3.3 Arcs in memory

In order to address this problem, we propose the 2-level hierarchy storage structure for a dependence graph. At the lower layer, arcs are stored in bit-packed storage efficient memory blocks that are intelligently and efficiently managed. The upper layer actually contains the unfolded version of the arcs stored at the lower layer. It usually keeps only a small number of arcs which are currently accessed in a active manner. The upper layer provides an interface between the lower layer and the users(the lower layer is invisible to them). The user does not have to worry about all the chores concerning low-level memory management since this hierarchy structure handles them in a uniform and clean way, including packing and unpacking of dependence arcs. The user can, therefore, take a high-level and abstract view of the dependence graph without losing performance in terms of space and time. This hierarchy structure is based on our observation that compared to the size of the whole dependence information, only a small portion of the information is likely to be accessed frequently for a short period of time. Since data dependence information is clustered by a loop-nest basis and loop transformation is performed loop-nest by loop-nest, it is mostly likely for the user to access a small part of arcs in a dependence graph for the time being.

A C++ class, **DDgraph**, take control of all these data and the access operations against them. One DDgraph class object is assigned to the **ProgramUnit** class in Polaris, and thus each DDgraph contains data dependence information for the whole subroutine or function. The arcs, called *inactive*, are stored in bit-packed memory blocks, **Arc_type**. Every inactive arc comprises the following fields: a pointer to the target statement of this arc, a pointer to target expression, a pointer to the next Arc_type and a data field. The data field is a series of several dependence direction vectors, involving the following fields:

- Active(1 bit) - status of this arc, active/inactive.
- Dependence Decision(1 bit) - assumed/proved dependence.
- Arc Direction(1 bit) - normal/reverse.
- Dependence Type(2 bits) - flow/anti/input/output dependence.
- Direction Vectors(3 bits a direction) - $<, >, =, \neq, \leq, \geq$, *, delimiter.

The Arc_types with the same source expression are singly linked all together and the source expression points to the head of the list. If two arcs has the same pair of expressions but with the different arc direction, that is, the source of this arc is the target of that arc, then they can be merged into a Arc_type list with an additional bit, Arc Direction. The DDgraph class prevents the user from directly accessing these inactive arcs. All the detail about the operations on these is hidden inside the DDgraph class.There already exist some compilers that try to optimize the memory usage of the dependence graph by packing the direction vectors invariably into one 32-bit word. The drawback of this structure is that we can only analyze loops nested at most 10 deep. As another problem with this, it must always allocate a word to store a direction vector even if we only need 3 bits for loops nested 1 deep. These problems are significant because in real application programs most of the loops are nested less than 3 to 4 and occasionally nested more than 10 deep. Since the Arc_type is not confined in fixed size, it always fits to optimal size without wasting memory space and also without any size restrictions. The overhead of expanding and shrinking the Arc_Type is effectively reduced by the upper layer structure that will be discussed below.

3.4 Active Arcs

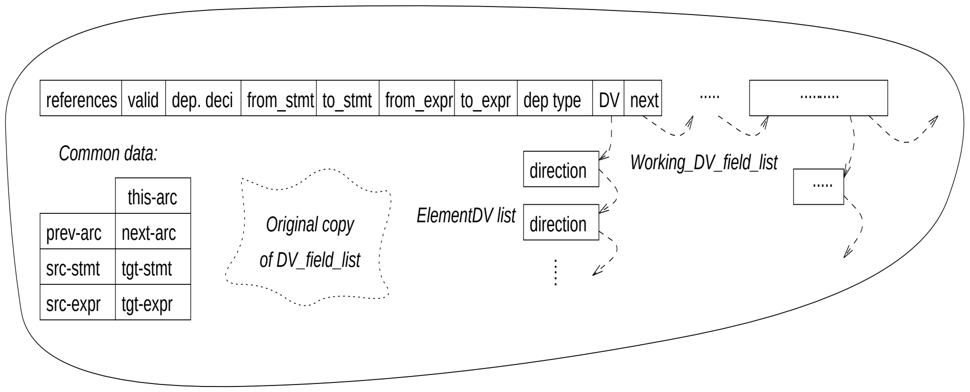

Figure 2: Active_Arc structure

Unlike the inactive arcs, the purpose of the *active* arc is different; a fast and flexible access. For this purpose, we provide another class, **Active_Arc**. In Figure **??**, the data structure of Active_Arc is shown. When the user tries to access some arc and the arc is currently inactive, the DDgraph unfolds the inactive arc and creates a new Active_Arc object to copy the unfolded information. The newly created Active_Arc is inserted into a Active_Arc list in the DDgraph. Each Active_Arc initially contains two copies of the corresponding Arc_type, that is, **Original_DV_field_list** and **Working_DV_field_list**. One is for the original archive, and the other is for the user access. If the user finishes all the operations on the arc and thus the active arc is no longer needed, then both the copies are compared. If both are different because the user changes the active arc information somewhere, then the original inactive arc is updated by the contents of the Working_DV_field_list. This Active_Arc structure is similar to that of a cache in the memory hierarchy system. The Active_Arc list in the DDgraph corresponds to a cache itself, and each Active_Arc corresponds to a cache line. We do not yet quantify the performance improvement due to this Active_Arc structure, but we speculate we reduce the overall time to access dependence graphs with this. The overall structure about the DDgraph and the other classes in Polaris is illustrated in Figure **??**.

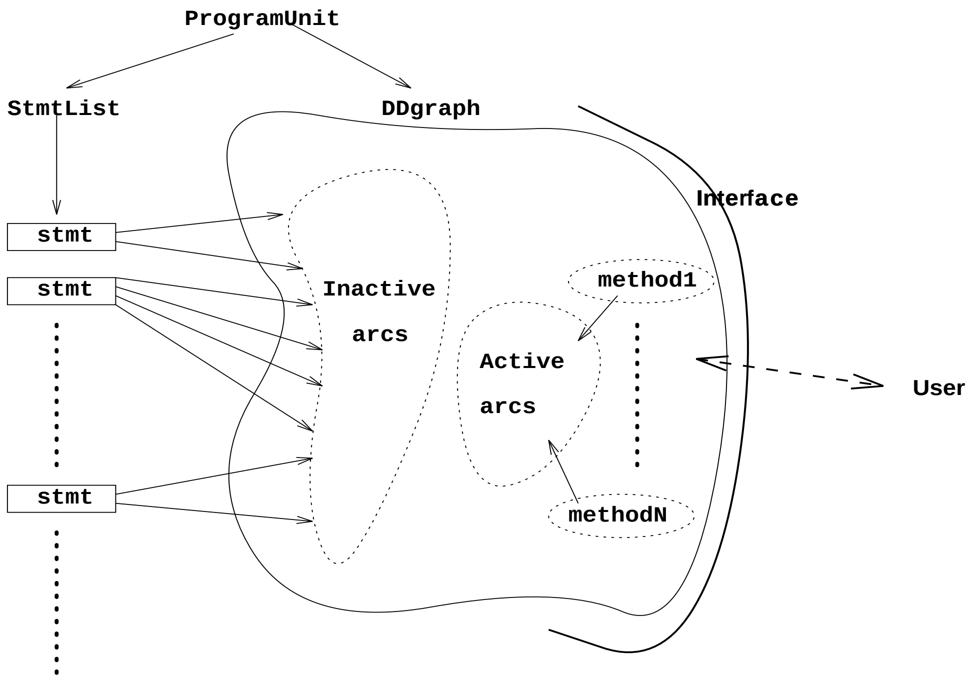

Figure 3: Overall structure of DDgraph

4 Assistance tools for DDgraph

4.1 DDiterator for DDgraph

Even though the Active_Arc class supports easier access, it still contains the complicated and physically detailed fields such as a Valid bit and a Reference counter. The user needs furthermore high-level view of a dependence graph. We have another class, **DDiterator**, to assuage this requirement. The DDiterator does not only provide high-level abstract operations, but also has a lot of flexibility about which subset of the arcs the user wants to look at, along with not requiring much overhead. The subset can be a pair set of arcs of lists of statements in a program, expressions or their combination. Each DDiterator object represents one subset and methods to access the dependence information for the subset. since the order in which the subset of the arcs is iterated over is automatically decided by the DDiterator, the user is simply requred to ask the direction of movements, that is, move to the previous or to the next. The major operations the user can use are as follows:

- modify(*new arc*) - substitute *new arc* for the current arc.
- del() - delete the current arc.
- grab() - delete the current arc and return the pointer to it.
- next(), prev() - move to the previous/next arc of the current arc.
- reset(), set_to_last() - set the current arc to the first/the last arc.
- valid(), end(), current_valid(), current_invalid() - check the validity of the current arc.
- current() - return a reference to the current arc.

4.2 Merging direction vectors

To abate excessive memory requisite for a data dependence graph, we use the two- level storage structure as described above, but we found more storage optimization is possible for redundant direction information. The DDgraph itself does not generate its arcs and other information, but instead in behalf of the user efficiently manage them once the user feed them into the DDgraph. Since the user might collect dependence information in several places, the directions collected by the user often contain redundant directions which can be merged into more general ones, resulting in memory waste. With a few number of directions, the merging job is simple and straightforward, but with a larger number of directions, the merging is not simple anymore and even the result can not be guaranteed to be correct. So, we provide a function **mergeDVs** that is called by the user with direction vectorss as input and that returns the merged and possibly less number of directions. The naive algorithm for merging the arbitrary number of directions has exponential time bound, but mergeDVs is a polynomial time algorithm. This polynomial time is enabled by bottom-up approach to derive more general directions from the subdirections. A simple example of merging direction vectors is illustrated in Figure **??**.

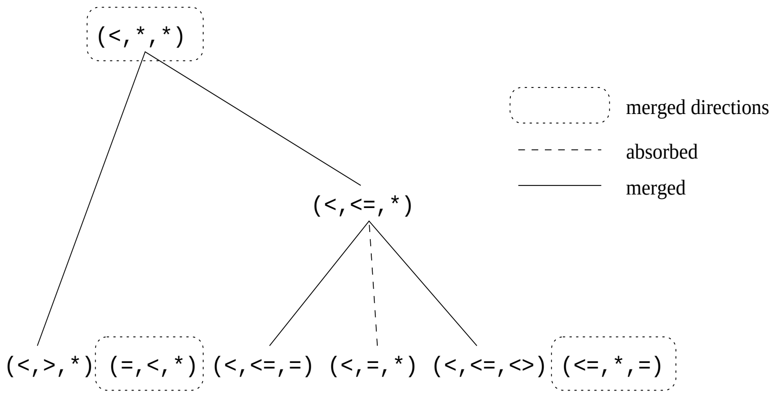

Figure 4: Direction merging

5 Future Direction

We still have room for improvement of the DDgraph and its functionality. First, we need other types of searching tools on the top of the DDiterator. For instance, currently, if the user wants to access the dependence information of some subset of the DDgraph, he must name all the statements belonging to the subset. Since we quite often perform a loop-base search, it is useful and convenient to provide some interface by which the user designates the outermost loop instead of enumerating all the statements inside of it. Second, We need distance information. Distance information obviously enables us to drive more aggressive and accurate parallelization. But, since the storage schemes for both types of information(a 3-bit pattern for a direction and an integer for a direction) are different, it is questionable how well the storage structure is organized to make both coexist efficiently in terms of access time and space usage. The DDgraph representation transparently allows you to intermix distance vectors with direction vectors. A bit in the inactive arc can tell us which is stored.

We do not yet compare the performance of our scheme to other ones such as the one with fixed sized inactive arcs and the one without active arc structure. We will evaluate how much space is needed and how fast the time is for every scheme to characterize data dependence graph usage in parallelizing compilers.

6 Conclusion

Data dependence information represented by a data dependence graph is essential through all the loop transformation passes in a parallelizing compiler. As we go through the passes, the requirement for the necessary information keeps changing pass by pass, and also the information must often be refined by the earlier passes for the use of the following ones. To support this, we need to store a data dependence graph in memory instead of generating it each time we need. This helps the compiler run faster. But, the new problems that arise here are how to organize the graph efficiently in memory, because most of the application programs need very big dependence graphs. The simple-minded storage scheme might save us significant space, but this is not the solution either. We need fast and feasible access to the stored information since the graph would be read and written back frequently by the user. So, besides a static structure which manage the whole graph in the very compact form, we also need a dynamic structure which provids fast and easy interface to the static structure. We here proposed two-level hierarchy storage scheme for this purpose. Since the dynamic structure is always kept as small as possible, we do not need much more memory for the additional structure. Dynamic structure guarantees the fast and easy access by using small-sized fully linked list structure, and asked by caching the dynamic(or active) arcs recently used, we can quickly respond to the reuse request for them which is most likely to occur by locality.

The usage of data dependence graph is broad. Loop vectorization, loop parallelization and loop distribution need this for checking $<$ or $=$ dependence directions in a loop. Loop interchange also capitalizes upon this to see if there is any dependence $(<,>)$ or $(>,<)$ in a loop. The DDgraph also generates input dependence that is not necessary in most of the loop transformation but very useful for locality enhancement techniques. In this case, the distance information is more useful than the direction only. These broad and various requirements for the data dependence graph account for the need for efficient parallelizing compilers.

References

[1] Wolfe M. and Banerjee U., Data Dependence and its Application to Parallel processing, *International Journal of Parallel Programming*, 16(2), pp. 137-178, Apr. 1987.

[2] Wolfe M., *Optimizing Supercompilers for Supercomputers*, The MIT Press, Cambridge, MA, 1989.

[3] Zima H. and Chapman B., Supercompilers for Parallel and Vector Computers, ACM Press, New York, NY, 1990.

[4] Huson C., An In-Line Subroutine Expander for Parafras, M.S. thesis 351, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Sci., Dec., 1982.

[5] Padua D. *et al*, Polaris: the next Generation in Parallelizing Compilers, Seventh Annual Workshop on Languages and Compilers for Parallel Computing, Cornell Univ., Ithaca, NY, pp. 10.1-10.18, Aug. 8-10.

[6] Blume W. and Eigenmann R., The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. Technical Report 1345, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & De., Apr. 1994.

[7] Rauchwerger L. and Padua D., The PRIVATIZING DOALL Test: A Run-Time Techniques for DOALL Loop Identification and Array Privatization. Technical Report 1329, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & De., Jan. 1994.

[8] Tu P. and Padua D., Automatic Array Privatization, Sixth Annual Workshop on Languages and Compilers for Parallel Computing, Portland, OR, pp. 500-521, Aug. 1993.

[9] Pugh W., The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis, *Supercomputing '91*, 1991

[10] Li Z., Yew P. and Zhu C., An Efficient Data Dependence Analysis for Parallelizing Compilers, *IEEE Trans. Parallel and Distrib. Syst.*, 1(1), pp. 26-34, 1990.

[11] Wolfe W. and Tseng C., The Power Test for Data Depedence, *IEEE Trans. on Parallel and Distrib. Syst.*, 3, pp. 591-601, Sep. 1992.

[12] Goff G., Kennedy K. and Tseng C., Practical Dependence Testing, *SIGPLAN Notices*, 26, pp. 15-29, June 1991.

[13] Kuck D. *et al*, The Perfect Benchmarks: Effective Performance Evaluation of Supercomputers, International Journal of Supercomputer Applications, 3(3), pp. 5-40, Fall 1989.

[14] Padua D. *et al*, The Polaris Internal Representation. Technical Report 1317. Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & De., Oct. 1993.