Combining compile-time and run-time parallelization [1]

Sungdo Moon [*], Byoungro So and Mary W. Hall

Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, CA 90292, USA

Tel.: +1 310 822 1510, ext. 458; Fax: +1 310 822 7791; E-mail: {sungdomo,bso,mhall}@isi.edu

This paper demonstrates that significant improvements to automatic parallelization technology require that existing systems be extended in two ways: (1) they must combine high-quality compile-time analysis with low-cost run-time testing; and (2) they must take control flow into account during analysis. We support this claim with the results of an experiment that measures the safety of parallelization at run time for loops left unparallelized by the Stanford SUIF compiler's automatic parallelization system. We present results of measurements on programs from two benchmark suites – SPECFP95 and NAS sample benchmarks – which identify inherently parallel loops in these programs that are missed by the compiler. We characterize remaining parallelization opportunities, and find that most of the loops require run-time testing, analysis of control flow, or some combination of the two. We present a new compile-time analysis technique that can be used to parallelize most of these remaining loops. This technique is designed to not only improve the results of compile-time parallelization, but also to produce low-cost, directed run-time tests that allow the system to defer binding of parallelization until run-time when safety cannot be proven statically. We call this approach *predicated array data-flow analysis.* We augment array data-flow analysis, which the compiler uses to identify independent and privatizable arrays, by associating predicates with array data-flow values. Predicated array data-flow analysis allows the compiler to derive "optimistic" data-flow values guarded by predicates; these predicates can be used to derive a run-time test guaranteeing the safety of parallelization.

Keywords: parallelizing compilers, shared-memory multiprocessors, data dependence analysis, array privatization, run-time parallelization

[1]This work has been supported by DARPA Contract DABT63-95-C-0118 and NSF Contract ACI-9721368.

[*]Corresponding author.

1. Introduction

Parallelizing compilers are becoming increasingly successful at exploiting coarse-grain parallelism in scientific computations, as evidenced by recent experimental results from both the Polaris system at University of Illinois and the Stanford SUIF compiler [3, 15]. While these results are impressive overall, some of the programs presented achieve little or no speedup when executed in parallel. This observation raises again questions that have been previously addressed by experiments in the early 90s [4,8,26]: Is the compiler exploiting all of the inherent parallelism in a set of programs, and if not, can we identify the techniques needed to exploit remaining parallelism opportunities?

These earlier experiments motivated researchers and developers of parallelizing compilers to begin incorporating techniques for locating coarse-grain parallelism, such as array privatization and interprocedural analysis, that have significantly enhanced the effectiveness of automatic parallelization. Now that the identified techniques are performed automatically by some compilers, it is an appropriate time to revisit these questions to determine whether further improvements are possible. This paper *empirically* evaluates the remaining parallelism opportunities using an automatic run-time parallelization testing system. Our approach is based on the Lazy Privatizing Doall (LPD) test, which tests whether a loop contains data dependences (different iterations access the same memory location, where at least one of the accesses is a write), and if so, whether such dependences can be safely eliminated with privatization (whereby each processor accesses a private copy of the data) [23]. For our system, we have defined and implemented the *extended-LPD test* (*ELPD*). ELPD extends LPD to test all loops in a loop nest simultaneously, rather than a single loop in a nest at a time, including when loop nests cross procedure boundaries. We use ELPD to instrument and test whether any of the candidate unparallelized loops in the program can be safely parallelized at run time. The implementation is based on the automatic parallelization system that is part of the Stanford SUIF compiler. We present measurements on programs from the SPECFP95 and NAS sample benchmarks.

The results of this experiment indicate that a new analysis technique, *predicated array data-flow analysis*, can be used to parallelize most of the remaining loops missed by the compile-time analysis in the SUIF compiler. This technique extends an existing implementation of array data-flow analysis by associating with each data-flow value a predicate; analysis interprets these predicate-value pairs as describing a relationship on the data-flow value when the predicate evaluates to true. A few existing techniques incorporate predicates, most notably guarded array data-flow analysis by Gu, Li, and Lee [10]. Our approach goes beyond previous work in several ways, but the most fundamental difference is the application of these predicates to derive run-time tests used to guard safe execution of parallelized versions of loops that the compiler cannot parallelize with static analysis alone. Run-time parallelization techniques that use an inspector/executor model to test all access expressions and decide if parallelization is safe can be applied to these same loops [23,24]. However, such techniques can potentially introduce too much space and time overhead to make them profitable. The run-time tests introduced by predicated analysis are, by comparison, much simpler.

Predicated array data-flow analysis unifies in a single analysis technique several different approaches that combine predicates with array data-flow values. By folding predicates into data-flow values, which we call *predicate embedding*, we can produce more precise data-flow values such as is achieved in the PIPS system by incorporating constraints derived from control-flow tests [18]. By deriving predicates from operations on the data-flow values, which we call *predicate extraction*, we can obtain breaking conditions on dependences and for privatization, such that if the conditions hold, the loop can be parallelized. The notion of breaking conditions has been suggested by Goff and by Pugh and Wonnacott [9,22]. We discuss how such conditions can be derived in much broader ways, and present how to use these conditions both to improve compile-time analysis or as the basis for run-time tests.

The remainder of the paper is organized into four main sections, related work and a conclusion. The next section presents background on the existing parallelization analysis in SUIF. We describe our instrumentation system and results of the instrumentation experiment in Section 3. The subsequent section describes predicated array data-flow analysis. Section 5 presents speedup measurements from applying predicated array data-flow analysis.

2. Background on parallelization analysis

The system described in this paper augments and extends an existing automatic parallelization system that is part of the Stanford SUIF compiler [14–16]. SUIF parallelizes loops whose iterations can be executed in parallel on different processors. To meet this criterion, the memory locations accessed by each iteration of a loop (and thus by each processor) must be *independent* of locations written by other iterations (and other processors). In some cases, accesses to a scalar or array variable within a loop must be transformed to make each processor's memory accesses independent. As one example, if all locations read by an iteration are first written within the same iteration, it may be possible to *privatize* the variable so that each processor accesses its own copy. Privatization is possible if there are no read accesses within an iteration of the loop that are *upwards exposed* to the beginning of the iteration; a read access is upwards exposed if there is a possible control flow path from the beginning of the loop body to the read access that contains no definition of the accessed location.

The compiler uses an interprocedural array data-flow analysis to determine which loops access independent memory locations, or for which privatization eliminates remaining dependences [16]. The analysis computes data-flow values for each program region, where a region is either a basic block, a loop body, a loop, a procedure call, or a procedure body. The data-flow value at each region consists of four component sets, $\langle$*Read, Exposed, Write, MustWrite*$\rangle$, defined as follows. *Read* describes the portions of arrays that may be read inside the program region. *Exposed* describes the portions of arrays that may have upwards exposed reads inside the program region (*Exposed* $\subseteq$ *Read*). *Write* describes the portions of arrays that may be written inside the program region. *MustWrite* describes the portions of arrays that must be written inside the program region (*MustWrite* $\subseteq$ *Write*). These four components consist of summaries to represent the array regions accessed within the program region. The summary for a given array contains a set of array regions, with each array region represented by a system of inequalities describing the constraints on the boundaries of the array region. A set of array regions is used instead of a single array region to avoid loss of information when multiple, very different accesses to an array appear in the same loop; in previous work, we have found this feature of the implementation to be very important to the precision of the result and have not found the size of the summaries to grow unmanageably.

We omit discussion of how these data-flow values are calculated, and refer the reader to previous publications [1,13,16]. At each loop, analysis tests whether there are arrays involved in data dependences, and if so, whether privatization is safe. If all dependences can be eliminated with privatization, the compiler determines that the loop is parallelizable. At a particular program region corresponding to loop L, the portions of arrays described by each component of the data-flow value are parameterized by loop index variable i (where, for clarity of presentation, i is assumed to be normalized to start at 1 and step by 1). Below we define the dependence and privatization tests for loop L, $\mathit{Independent}_L$ and $\mathit{Privatizable}_L$. The notation $\mathit{Write}_L|_i^{i_1}$ refers to replacing i with some other index i_1 in the iteration space.

$$\mathit{Independent}_L \Leftrightarrow$$

$$\forall i_1, i_2 \in I,\ i_1 \neq i_2,\ \left(\mathit{Write}_L|_i^{i_1} \cap \mathit{Read}_L|_i^{i_2} = \emptyset\right) \wedge \left(\mathit{Write}_L|_i^{i_1} \cap \mathit{Write}_L|_i^{i_2} = \emptyset\right)$$

$$\mathit{Privatizable}_L \Leftrightarrow$$

$$\forall i_1, i_2 \in I,\ i_1 \neq i_2,\ \left(\mathit{Write}_L|_i^{i_1} \cap \mathit{Exposed}_L|_i^{i_2} = \emptyset\right)$$

An array is involved in a data dependence if: (1) the same location is written in some iteration i_1 and is read in some other iteration i_2 (a true or anti-dependence); or, (2) the same location is written in two different iterations i_1 and i_2 (an output dependence). An array is not privatizable if the same location is written in some iteration i_1 and is upwards exposed to the beginning of the loop in some other iteration i_2. This test for privatization permits exposed reads for some of the array locations, but only if they are either not written in the loop or are written in the same iteration as the read. To correctly privatize such arrays requires that the compiler initialize the upwards exposed locations in the private copy of the array prior to executing the loop.

3. Instrumentation

The previously described dependence and privatization tests must err on the conservative side. In cases where array subscript expressions are too complex for the compiler to analyze, the compiler may report a dependence and that privatization is not possible when, in fact, the loop can be safely parallelized. The purpose of the instrumentation system is to test at run-time whether parallelization is safe for all candidate loops that the compiler failed to parallelize statically.

At each loop, the instrumentation system performs a run-time dependence and privatization test based on the Lazy Privatizating Doall (LPD) test [23]. We could not use the LPD test because, as it is defined, it only instruments a single loop in a loop nest. A loop nest might contain several loops left unparallelized by the compiler. Because our goal is to pinpoint all the candidate parallelizable loops in a nest, we need to know which of the unparallelized loops in a nest is inherently parallel. For this purpose, we have defined and implemented the *extended-LPD test* (*ELPD*). ELPD extends LPD to test all loops in a loop nest simultaneously, including when loop nests cross procedure boundaries. Using ELPD, we are thus able to locate all the loops in the program whose iterations can be safely executed in parallel for a particular program input, possibly requiring array privatization to create private copies of an array for each processor.

We now describe the ELPD test, which reformulates the previously defined dependence and privatization tests, and present the results of applying the instrumentation system to the benchmark programs.

3.1. ELPD (extended-LPD) test

The instrumentation system uses the results of array data-flow analysis and dependence and privatization tests to decide which loops and which variables in each loop should be instrumented. An initial instrumentation analysis phase designates loops and arrays for instrumentation. The system reduces the amount of work performed by not instrumenting loops nested inside already parallelized loops. This feature is important because the SUIF run-time system only exploits a single level of parallelism, so the inner loop would not be parallelized even if proven to be parallel. The results of the instrumentation analysis phase are two sets at each loop L, *Instr(L)* and *GInstr(L)*. *Instr(L)* is the set of arrays in L that must be instrumented to test for dependence and privatization within L. *GInstr(L)* is the set of arrays accessed in L that must be instrumented to test for dependence and privatization in some outer enclosing loop (i.e., globally requiring instrumentation).

A detailed explanation of the instrumentation analysis phase can be found in [27].

A subsequent transformation phase actually inserts the instrumentation code. The instrumentation analysis is fully interprocedural, and it is implemented in the interprocedural framework that is part of the SUIF system [13,16].

We begin by presenting how instrumentation is performed within a single loop. For some array in *Instr*(L), with dimension sizes specified as $A[1:d_1, 1:d_2, \ldots, 1:d_n]$, for loop L with bound $b = [1:b_u]$, the system introduces four shadow arrays: S_w (marks elements written within L), S_r (marks elements read but not written within at least one iteration of L), S_{np} (marks elements that are read only or read before written within at least one iteration of L, for use in the privatization test), and S_{rf} (marks elements read first before any writes for all of L). These arrays are of the same dimensionality as A but are integer or boolean only. We also introduce a single boolean O that is set if the loop contains an output dependence.

Let $I = (i_1, i_2, \ldots, i_n)$ refer to a subscript expression for array A appearing in the code for loop L, which we call an access function. To clarify the algorithms presented below, we first define the shadow arrays and a set of properties of the values of shadow array elements for the location described by access function I at completion of loop L's execution.

- $S_w[I] = b \Leftrightarrow b$ is the last iteration of L that writes location $A[I]$.
- $S_r[I] = b \Leftrightarrow b$ is the first iteration of L that reads and does not write location $A[I]$.
- $S_{np}[I] = \mathit{true} \Leftrightarrow A[I]$ has an upwards exposed read in some iteration of L.
- $S_{rf}[I] = \mathit{true} \Leftrightarrow$ the first access to $A[I]$ in L is a read.
- $O = \mathit{true} \Leftrightarrow$ write accesses to some location $A[I]$ occur in different iterations of L.

The shadow array elements for S_r and S_w are initialized to 0. The elements for S_{np} and S_{rf}, and boolean O, are initialized to false.

During program execution, the system performs the following added computations in response to read and write accesses of A. The term b_c refers to the current iteration of loop L, and b_a indicates any iteration value $\neq 0$.

Write $A[I]$

```
**if** $(S_w[I] \neq b_c)$
    **if** $(S_w[I] \neq 0)$ $O = \text{true}$
    **if** $(S_r[I] = b_c)$ $S_r[I] = 0$
    $S_w[I] = b_c$
```

Read $A[I]$

```
**if** $(S_w[I] \neq b_c)$
    **if** $(S_r[I] = 0)$ $S_r[I] = b_c$
    **if** $(S_w[I] = 0)$ $S_{rf}[I] = \mathit{true}$
    $S_{np}[I] = \mathit{true}$
```

Upon exit of the loop, the ELPD test examines the shadow arrays to determine whether accesses to the array are independent, or if not, whether dependences can be safely eliminated with privatization. We now show how to reformulate the dependence and privatization tests from Section 2 as run-time tests on the shadow arrays, for a particular array accessed in L.

The dependence test is defined as follows.

$$\mathit{Independent}_L \Leftrightarrow$$

$$(\forall I \in [1:d_1, 1:d_2, \ldots, 1:d_n],$$
$$(S_r[I] = 0) \vee (S_w[I] = 0)) \wedge (O = \mathit{false})$$

The first term of the dependence test determines whether there are loop-carried true or anti-dependences, while the second term determines whether there are loop-carried output dependences. If there is an output dependence but no true or anti-dependence, we can apply the following test to determine whether the output dependence could be eliminated by privatization.

$$\mathit{Privatizable}_L \Leftrightarrow$$

$$\forall I \in [1:d_1, 1:d_2, \ldots, 1:d_n],$$
$$(S_{np}[I] = \mathit{false}) \vee (S_w[I] = 0)$$

Given that we have already proven that there is an output dependence and no true or anti-dependences, the privatization test determines whether there is a read upwards exposed to the beginning of some iteration that is also written in some other iteration of the loop.

As compared to the LPD test, this formulation of the single-loop ELPD test introduces the additional shadow array S_{rf} to recognize whether the first access to an array element in a loop is a read. This shadow array is needed only to derive the solution at an outer loop based on accesses in the inner loop, as described below. This shadow array can be omitted for the outermost loop in a nest for which the ELPD test is performed.

A distinguishing feature of our system is that it may instrument multiple loops in a nest. Suppose loop L_k is nested inside loop L_j and for both loops, array A must be instrumented (i.e., $A \in \mathit{Instr}(L_k) \wedge A \in \mathit{Instr}(L_j)$). Then each loop will have its own shadow arrays for array A. In this case, we need to update the values of the shadow arrays of the outer loop following completion of each invocation of the inner loop. Given shadow arrays S^k_w, S^k_r, and S^k_{rf} for the inner loop and shadow arrays S^j_w, S^j_r, S^j_{np}, and S^j_{rf}, and boolean O^j for the outer loop, the updated values for the outer loop's shadow arrays is defined below. (When shadow arrays for the outer loop appear on the right hand side of equations below, we are referring to the value prior to performing these updates.)

$$O^j = \begin{cases} \mathit{true} & \text{if } (S^j_w[I] = b_a) \wedge \\ & \quad (S^k_w[I] \neq 0) \wedge (b_a \neq b^j_c) \\ O^j & \text{otherwise} \end{cases}$$

$$S^j_w[I] = \begin{cases} b^j_c & \text{if } (S^k_w[I] \neq 0) \\ S^j_w[I] & \text{otherwise} \end{cases}$$

$$S^j_r[I] = \begin{cases} 0 & \text{if } (S^j_r[I] = b^j_c) \wedge \\ & \quad (S^k_w[I] \neq 0) \\ b^j_c & \text{if } (S^j_r[I] = 0) \wedge \\ & \quad (S^j_w[I] \neq b^j_c) \wedge (S^k_r[I] \neq 0) \wedge \\ & \quad (S^k_w[I] = 0) \\ S^j_r[I] & \text{otherwise} \end{cases}$$

$$S^j_{np}[I] = \begin{cases} \mathit{true} & \text{if } (S^j_w[I] \neq b^j_c) \wedge \\ & \quad (S^k_{rf}[I] = \mathit{true}) \\ S^j_{np}[I] & \text{otherwise} \end{cases}$$

$$S^j_{rf}[I] = \begin{cases} \mathit{true} & \text{if } (S^j_w[I] = 0) \wedge \\ & \quad (S^k_{rf}[I] = \mathit{true}) \\ S^j_{rf}[I] & \text{otherwise} \end{cases}$$

The shadow array $S^j_w[I]$ sets its iteration value to the current iteration if it is written in the inner loop. The shadow array $S^j_r[I]$ sets its value to the current iteration if it was previously unread in the outer loop, was not written in the current iteration of the outer loop and was read but not written in the inner loop; its value is set to 0 if it was previously the current iteration, and it is written by the inner loop. The shadow array $S^j_{np}[I]$'s value is only set to true if it was not written in the current iteration of the outer loop, and is read in the inner loop before possibly being written. Similarly, $S^j_{rf}[I]$ is set to true if it has never been written in the outer loop and is read in the inner loop before possibly being written.

In general, this computation is necessary whenever $A \in \mathit{GInstr}(L) \wedge A \in \mathit{Instr}(L)$. The global arrays to be updated are the shadow arrays either from the immediately enclosing loop or, if L is the outermost instrumented loop in the procedure, the shadow arrays passed into the procedure as parameters. In the latter case, the index of the loop from the invoking procedure is also passed as a parameter.

3.2. Instrumentation results

We have evaluated the instrumentation system on the SPECFP95 and NAS sample benchmark suites. We used reference inputs for SPECFP95 and the small inputs for NAS. One program was omitted from our results, fpppp, due to non-standard Fortran that our compiler does not accept.

Our experiments consider two benchmark suites for which the SUIF compiler was already mostly successful at achieving good speedups. In a previous publication, SUIF achieved a speedup on seven of the SPECFP95 programs; of these seven, su2cor achieved a speedup of only 4 on 8 processors of a Digital Alphaserver 8400 [15]. The remaining six obtained a speedup of more than 6. The programs apsi, wave5 and fpppp were the only three not to obtain a speedup. In the NAS benchmark suite, only buk and fftpde failed to achieve a speedup. To obtain the results presented below, we executed the instrumented program to locate the ELPD-proven parallel loops.

Collectively, these 17 programs contain almost 2000 loops, and the base SUIF system parallelizes over 70% of them [1]. (Note that the base SUIF system used in this paper parallelizes a few more loops than in previous work.) Of the remaining roughly 500 loops, some are not candidates for parallelization because of read I/O or internal exits or because they are nested inside already parallelized loops; others do not execute at run time. The system instruments 176 candidate loops that execute at run time and have more than a single iteration per invocation. Of these, just 59 are found to be parallel by the ELPD test.

From the results, we see that overall the compiler was already doing a good job of parallelizing these applications. Only eight of the seventeen programs (where mgrid and applu are counted twice) contain ELPD-proven parallelizable loops that the compiler missed. Once we identified the programs with remaining parallelism opportunities, we examined the ELPD-proven parallelizable loops in these programs to evaluate how a parallelizing compiler might exploit this

Table 1

Requirements of remaining parallel loops

Program	RA	CF	BC	CF+BC	OD	IE	DD	Total
apsi	1	10	0	6	0	0	0	17
mgrid	0	0	0	0	0	1	0	1
su2cor	0	9	5	0	4	1	0	19
wave5	0	1	9	2	1	0	2	15
buk	0	0	0	0	0	1	0	1
cgm	0	0	0	0	2	0	0	2
fftpde	0	3	0	0	0	0	0	3
mgrid	0	0	0	0	0	1	0	1
Total	1	23	14	8	7	4	2	59

additional parallelism automatically. We characterized the requirements of these additional loops as presented in Table 1[2].

In the table, the programs are listed in the first column, with counts of additional inherently parallel loops appearing in the final column. The remaining seven columns provide a count of how many of the loops could potentially be parallelized with a particular technique. These techniques and requirements are defined as follows:

- **RA:** Identifies loops for which simple range analysis would permit the loop to be parallelized [5].
- **CF:** Identifies loops for which parallelization analysis fails because of control flow within the loop. The control flow paths that would result in a dependence can potentially be ruled out at compile time by associating predicates with array data-flow values during analysis and comparing predicates on reads and writes to rule out impossible control flow paths. While not in common practice, a few techniques refine their array data-flow analysis results in this way [10,30].
- **BC:** Identifies certain loops whose safe parallelization depends on values of variables not known at compile time. For the loops in this category, it is straightforward to derive *breaking conditions* by extracting constraints on dependences directly from the dependence and privatization tests [9,22].
- **CF+BC:** Identifies loops that require both breaking conditions and analysis taking control flow into account. In some cases, derivation of breaking conditions is not as simple as extracting them directly from the dependence test.
- **OD:** Identifies loops with only output dependences, for which SUIF's analysis was unable to determine how to finalize their values. The loops are parallelizable with some run-time assistance to determine the iteration of the last write of each location.
- **IE:** Identifies loops that can probably only be parallelized with an inspector/executor model such as LPD [23]. These loops contain potential dependences on arrays with subscript expressions that include other arrays (i.e., index arrays).
- **DD:** Identifies loops where an inspector/executor model is probably not even suitable, because they contain dependences that occur only under certain control flow paths through the loop, and the control flow tests are based on loop-varying variables within the loop. The only approach we know that could parallelize such loops is a speculative inspector/executor model, where the loop is parallelized speculatively, and the inspector is run concurrently with executing the loop [23].

The eight programs contain a total of 59 additional parallelizable loops found by the ELPD test. (Note that this number contains only loops that were executed at run time.) The two programs apsi and su2cor have the most loops, 17 and 19 loops. The keys to parallelizing apsi are to take control flow tests into account during analysis and derive simple run-time tests. Several of the loops in the **CF** column have compiler-assumed dependences on scalar variables only. The bulk of the large loops in su2cor can be parallelized by taking control flow into account. Wave5 has two large loops that require analysis that incorporates control-flow tests and introduces some run-time testing. The NAS program fftpde has large loops that can be par-

[2]The difference in these results as compared to a previously published version is mostly due to eliminating from the count those loops that only execute a single iteration. Parallelizing such loops is obviously not going to improve performance, and counting them skews the results. Also, we have made a few corrections.

allelized by taking control flow into account, but they also have complex nonlinear subscript expressions.

Overall, we see that most of the loops require some sort of run-time testing to verify the safety of parallelization, at least 35 of the 59 loops (all categories except **RA** and **CF**). But rather than always reverting to a potentially expensive inspector/executor model, we see that in 29 of the 35 loops requiring run-time testing, a less expensive and more directed run-time test can potentially be derived with other techniques (all categories requiring run-time tests other than **IE** and **DD**). We also observe from the table that taking control flow tests into account in analysis is very important, required for 31 of the 59 loops (in the **CF** and **CF+BC** categories). These results indicate that there is still some room for improvement in automatic parallelization in two areas: incorporating control flow tests into analysis and extracting low-cost run-time tests wherever applicable instead of using an inspector/executor.

4. Predicated array data-flow analysis

In this section, we present an overview of predicated array data-flow analysis. A more complete treatment is found elsewhere [19,20]. This technique can be used to parallelize the 45 loops that fall into the **CF**, **BC** and **CF+BC** categories from the experiment in the previous section.

4.1. Extending traditional data-flow analysis

Traditional program analysis computing a *meet-over-all-paths* (*MOP*) solution produces data-flow values that conservatively approximate the true data flow. A MOP solution assumes that all possible control-flow paths through a program may be taken; at points of confluence where separate control-flow paths merge, a *meet* function is applied to the data-flow values representing the different paths to derive a single data-flow value that conservatively approximates the values for both paths. Also, analysis must derive data-flow values that conservatively approximate analysis results for all possible program inputs. Predicated data-flow analysis, instead, produces "optimistic" data-flow values guarded by predicates. Dependence and privatization tests on predicated data-flow values lead to increased parallel loops in two ways: through improved compile-time analysis, and by enabling analysis to derive run-time parallelization and privatization tests.

To make this argument more concrete, we present four examples in Fig.1. Fig. 1(a) shows an example that could be parallelized by previous array data-flow analysis techniques incorporating predicates [11,29]. In Fig. 1(a), traditional data-flow analysis determines that there may be an upwards-exposed use of array *help* because there is a possible control flow path through the loop that references *help* but bypasses the preceding assignment to it. Predicated data-flow analysis discovers that the predicates for the assignment and reference of y are equivalent; thus, none of array *help* is upwards exposed.

The next three examples require additional features of predicated array data-flow analysis, which we summarize here and discuss in more detail below. Fig. 1(c) shows how predicate embedding can also be used to improve the results of compile-time analysis. Run-time tests are required for safe parallelization in the examples in Figs. 1(b) and (d); we show how our analysis derives run-time tests to parallelize more loops than these previous approaches.

In Fig. 1(b), *help* is upwards exposed for certain values of x. Deriving predicates during dependence and privatization testing on predicated data-flow values leads to the appropriate run-time test for this loop. First, we consider whether a dependence exists on *help*, which occurs if there is an intersection of read and write regions and their predicates both hold. Thus, there is a dependence only if both $x > 2$ and $x > 5$ hold, which is equivalent to $x > 5$. Thus, if *NOT*($x >$ 5), the loop can be parallelized as written. Second, we compare the upwards exposed read regions and the write regions to determine if privatization is safe and discover that array *help* is privatizable if $x > 5$, the only condition where an upwards exposed read intersects with a write. These two cases enable the compiler to parallelize all possible executions of the loop.

In examples such as in Fig. 1(c), most compilers would assume that the element *help*[0] is upwards exposed because the loop assigns to only *help*[$1 : d$] but it possibly references *help*[$j - 1$] and j ranges from 1 to d. Through predicate embedding, the compiler can use the constraint $j > 1$ from the control-flow predicate inside the else branch of the second j loop to prove that *help*[0] is not accessed by the loop, and, as a result, *help* can be safely privatized and the loop parallelized.

In Fig. 1(d), *help*[1] may be upwards exposed if $d < 2$, since in this case the first loop containing the writes to *help* would not execute. Predicate extraction can be used to derive the condition $d < 2$ as a run-time test to determine whether to privatize *help* or leave it as written. Both versions of the loop can then be parallelized.

```
for i = 1 to c
    for j = 1 to d
        if (x > 5) then
            help[j] = ...
    endfor
    for j = 1 to d
        if (x > 5) then
            ... = help[j]
    endfor
endfor
```

(a) improves compile-time analysis

```
for i = 1 to c
    for j = 1 to d
        if (x > 5) then
            help[j] = ...
    endfor
    for j = 1 to d
        if (x > 2) then
            ... = help[j]
        endfor
endfor
```

(b) derives run-time test

```
for i = 1 to c
    for j = 1 to d
        help[j] = ...
    endfor
    for j = 1 to d
        if (j = 1) then
            ... = help[j]
        else
            ... = help[j-1]
    endfor
endfor
```

(c) benefits from predicate embedding

```
for i = 1 to c
    for j = 2 to d
        help[j-1] = ...
        help[j] = ...
    endfor
    for j = 1 to d
        ... = help[j]
    endfor
endfor
```

(d) benefits from predicate extraction

Fig. 1. Examples benefitting from predicated analysis.

4.2. Description of technique

The above presentation describes features of a prototype implementation of predicated array data-flow analysis that is part of the Stanford SUIF compiler. While space considerations preclude a formal description of predicated array data-flow analysis, we touch on what modifications to an existing array data-flow analysis are required to realize this solution. The technique is described in more detail elsewhere [19,20].

1. *Analysis augments array data-flow values with predicates.* Array data-flow analysis in the base SUIF compiler maintains, for a particular array, a *set* of array regions (instead of a single region that conservatively approximates all of the accesses within a loop). Since SUIF is already maintaining multiple array regions per array, it is straightforward to extend each array region to have an associated predicate. Depending on the data-flow problem being solved, this predicate is interpreted as being conservative towards true (for union problems such as *Read*, *Write* and *Exposed*) or conservative towards false (for intersection problems such as *MustWrite*). That is, for union problems, analysis errs towards assuming a value holds, while for intersection problems, analysis errs towards assuming a value cannot hold.
2. *Analysis redefines key operators to correctly support predicated array data-flow values.* At a given program point, calculating upwards exposed read regions involves a subtraction of the *Exposed* regions of a body of code (such as a basic block) with the *MustWrite* regions of the preceeding body of code. The subtraction operator, as well as intersection and union operators (the meet functions for *MustWrite*, and the other data-flow problems, respectively) have been redefined for predicated array data-flow analysis. All other operators remain unchanged.
3. *The system modifies dependence and privatization tests.* The dependence and privatization tests described in Section 2 that return just true or false must be extended to derive as solutions run-time parallelization tests that can be used to guard execution of a parallelized version of the loop. Fig. 2 presents the predicated versions of the dependence and privatization tests. We introduce

$$\mathit{Intersect}(r,s) = \exists i_1, i_2 \in I \mid (i_1 \neq i_2) \wedge \left(r|_i^{i_1} \cap s|_i^{i_2} \neq \emptyset\right)$$

$$\mathit{VPS}(r,S) = \begin{cases} \mathit{true}, & \mathit{if}\, \forall \langle p,s\rangle \in S,\ \mathit{Intersect}(r,s) = \mathit{false} \\ \neg\left(\bigvee_{\langle p,s\rangle \in S \wedge \mathit{Intersect}(r,s)} p\right), & \mathit{otherwise} \end{cases}$$

$$\mathit{Independent}_L = \bigwedge_{\langle p,r\rangle \in R_L} (\neg p \vee \mathit{VPS}(r, W_L)) \wedge \bigwedge_{\langle p,w\rangle \in W_L} (\neg p \vee \mathit{VPS}(w, W_L))$$

$$\mathit{Privatizable}_L = \bigwedge_{\langle p,e\rangle \in E_L} (\neg p \vee \mathit{VPS}(e, W_L))$$

$$\mathit{Initialize}_L = \begin{cases} \bigcup_{\langle p,e\rangle \in E_L} \langle p,e\rangle, & \mathit{if}\, \mathit{Privatizable}_L \neq \mathit{false} \\ \emptyset, & \mathit{otherwise} \end{cases}$$

Fig. 2. Dependence and privatization test on predicated data-flow values.

a test *Intersect* to determine if a region intersects another region on different iterations. When modifying the tests for predicated array data-flow analysis, we make use of the *Value-to-Predicate Solution*, ($\mathit{VPS}(r,S)$) to compare a data-flow set S to a single array region r and provide a predicate guaranteeing that r does not intersect with any array regions in S. This predicate is derived from the predicates on all regions that intersect with r. If $\mathit{VPS}(r,S)$ returns false, the analysis must assume that r always intersects with the array regions in S. If $\mathit{VPS}(r,S)$ returns true, the compiler has proven r never intersects with the array regions in S. If neither true nor false, the result corresponds to a run-time evaluable predicate that may be either compared with other predicates at compile time or tested at run time. To improve the precision of the VPS solution, we may derive additional predicates in the form of breaking conditions from the constraints arising from intersecting two regions. The conjunction of these breaking conditions with the predicate on the region in S refines the solution.
The dependence test uses the VPS solution to determine whether predicates guarding reads and writes of an array can be true simultaneously. The modified privatization test is similar. The initialization computation is augmented to return the set of $\langle p,e\rangle$ elements from exposed reads where privatization for the current array is possible; the result region must be initialized if predicate p holds.

5. Experimental results

In this section, we summarize a series of experimental results to measure the impact of predicated array data-flow analysis at finding additional parallelism. We applied our prototype implementation to programs in the SPECFP95 benchmark suite. Previously published results on this benchmark suite demonstrated speedups on 7 of the 10 programs. For 6 of these 7 programs, the speedup was 6 or better on an 8-processor Digital AlphaServer 8400, but the program su2cor obtained a speedup of less than 4. The three programs that did not speed up were apsi, fpppp, and wave5. We have identified loops in three of the four programs for which speedup was previously limited (see discussion in Section 3, apsi, su2cor, and wave5, that would benefit from predicated array data-flow analysis).

As one measure of the value of predicated data-flow analysis, we first consider how many more of the loops in our benchmark programs with *inherent loop-level parallelism* can be parallelized. By inherent loop-level parallelism, we refer to loops that either have no data dependences at run time, or for which privatization eliminates remaining dependences. There are 59 remaining inherently parallel loops in SPECFP95 and NAS found by using the ELPD test, as shown in Table 1.

Of these 59 parallelizable loops that were missed by the previous SUIF system, our predicated array data-flow analysis implementation parallelizes 35 of them, across 3 programs, as shown in Table 2. In the table, the first column gives the subroutine and line number for the loop. The second column provides the number of lines of code in the loop body. All of these loops span multiple procedure boundaries, so this line count is the sum of the number of lines of code in each procedure body invoked inside the loop. Even if a procedure is invoked more than once in the loop, it is counted only once. The column labeled Coverage is a measure of the percentage of sequential execution time of the program spent in this loop body. The column labeled

Table 2

Additional loops parallelized by predicated analysis

Loop	# of Lines	Coverage	Granularity	Category	Requirement
apsi					
run-909	1288	4.32	10.02	CF+BC	$\Leftarrow$,R
run-953	1288	4.31	9.99	CF+BC	$\Leftarrow$,R
run-1015	1288	4.31	9.98	CF+BC	$\Leftarrow$,R
run-1044	1218	3.23	7.48	CF+BC	$\Leftarrow$,R
run-1083	1287	4.31	10.00	CF+BC	$\Leftarrow$,R
run-1155	1268	3.47	8.03	CF+BC	$\Leftarrow$,R
dcdtz-1331	235	6.96	16.12	CF	$\Rightarrow$
dtdtz-1448	281	10.43	24.17	CF	$\Rightarrow$
dudtz-1642	258	10.24	23.72	CF	$\Rightarrow$
dvdtz-1784	261	9.86	22.83	CF	$\Rightarrow$
setall-4128	14	0.0005	1.21	CF(scalar)	
setall-4130	10			CF(scalar)	
topo-4539	30	0.0006	0.64	CF(scalar)	
dkzmh-6265	218	7.58	17.55	CF	$\Leftarrow$,R
su2cor					
sweep-420	237	28.97	22.88	CF	$\Leftarrow$
loops-1557	185	0.89	100.21	CF	$\Leftarrow$
loops-1558	184			CF	$\Leftarrow$
loops-1559	183			CF	$\Leftarrow$
loops-1613	265	2.61	292.67	CF	$\Leftarrow$
loops-1614	264			CF	$\Leftarrow$
loops-1659	573	22.46	2522.48	CF	$\Leftarrow$
loops-1660	572			CF	$\Leftarrow$
loops-1661	571			CF	$\Leftarrow$
trngv-2182	3	0.15	0.0098	BC	R
trngv1-2266	3	0.0011	0.0085	BC	R
wave5					
field-3087	4	0.0018	0.097	BC	R
field-3118	4	0.0016	0.086	BC	R
field-3367	26	0.42	22.91	BC	R
field-3396	4	0.0013	0.072	BC	R
field-3420	27	0.47	25.80	BC	R
field-3450	4	0.0011	0.061	BC	R
field-3465	25	0.34	18.39	CF	
field-3493	4	0.0005	0.025	BC	R
fftf-5064	1154	1.50	16.38	CF+BC	$\Leftarrow$,R
fftb-5082	1147	2.27	24.69	CF+BC	$\Leftarrow$,R

Granularity provides a per-invocation measure of the loop in milliseconds, indicating the granularity of the parallelism. (In our experience, granularities on the order of a millisecond are high enough to yield speedup.) Both coverage and granularity results were obtained on a single processor of a 195 MHz SGI Origin 2000. The next column provides a category for the loop corresponding to classifications defined in Section 3. The final column describes what components of the analysis are needed to parallelize each loop. The symbols $\Rightarrow$ and $\Leftarrow$ refer to whether predicate embedding or extraction are required, and R refers to loops that are only parallelizable under certain run-time values and require a run-time test.

In performing these experiments, we encountered some limitations in the base SUIF system that interfered with our analysis, most notably the interface between the symbolic analysis and the array data-flow

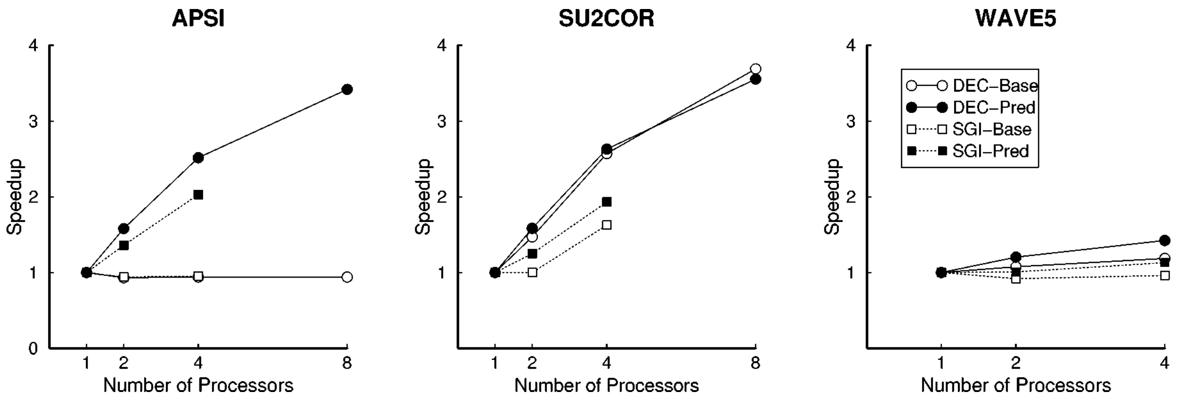


Fig. 3. Speedups due to predicated array data-flow analysis.

analysis. To focus on issues specific to predicated data-flow analysis, we performed a few transformations by hand to parallelize some of the loops. The second group of eight loops in apsi were parallelized completely automatically with no transformations to the code. The first six loops in apsi and the last two loops in wave5 required transformations to the original source before performing our analysis: forward substitution and cloning and loop peeling to enable forward substitution. The loops in su2cor were parallelized completely automatically.

We point to a few key results in the table. Embedding, extraction, and run-time tests, distinguishing features of our analysis, are common requirements for these programs. At least one is needed in 31 of the 35 loops. Further, many of the loops parallelized by predicated array data-flow analysis have high granularity and high coverage, and thus have the potential to yield speedup improvements on a moderate-scale multiprocessor system.

For these three programs with additional parallel loops, we measured the speedup on 4 processors of the SGI Origin as well as an 8-processor 300 MHz Digital AlphaServer 8400, as presented in Fig. 3. Each graph contains four lines. For each of the two machines, there is a line for the base SUIF system and a line for the predicated analysis version. Speedups are compared against a sequential version of the program. For those requiring a run-time test, we produced the parallel version using a user tool and modifications to the final output of the SUIF compiler; that is, the analysis is automated but the generation of conditionally parallel code is not.

On the DEC system, we obtained solid speedups for the first two programs and more modest speedups for wave5. On the SGI, we see that the speedups are not nearly as high as on the DEC, but predicated analysis yields improvements on every program. Su2cor does not yield an improvement on the DEC due to a more restrictive stacksize limitation than the SGI that requires us to allocate privatized arrays on the heap rather than on the stack.

Certainly, the most dramatic results are from apsi, which contains 14 loops benefitting from predicated array data-flow analysis, comprising roughly 70% of the program's execution time. Most of the loops cross multiple procedure boundaries, and the first six are the largest loops ever parallelized by the SUIF compiler. Parallelizing these additional loops translates to substantial improvement in speedup for the program, while without parallelizing these loops, the 4-processor parallel program does not speed up at all.

The compiler finds 11 additional loops to parallelize in su2cor, comprising 55% of the program's execution time. During experiments, we sequentialize the first loop *sweep-420*. While the *sweep-420* loop is reasonably coarse-grained and makes up over 28% of the program's execution time, parallelizing it actually slightly degrades overall speedup of the program because it executes only a few iterations and has a load imbalance problem. Moreover, loops nested inside *sweep-420* loop that are parallelized by the base SUIF compiler, scale much better. These problems could be mitigated if the SUIF run-time system exploited multiple levels of parallelism in a loop nest, but in the current system, it is more cost-effective to execute the inner loops nested inside of this one in parallel. (The current SUIF system can be configured not to execute such a loop because of its small number of iterations.) Note that the real benefit of predicated analysis for su2cor is in parallelizing the fourth loop, which has a extremely large granularity of 2.5 seconds per invocation.

On wave5, improvements are more modest because the loops parallelized by predicated analysis comprise only about 4% of the program's execution time. Nevertheless, the program goes from little or no speedup to modest speedups on both systems.

6. Related work

A number of experiments in the early 90s performed hand parallelization of benchmark programs to identify opportunities to improve the effectiveness of parallelizing compilers [4,8,26]. These experiments compared hand-parallelized programs to compiler-parallelized versions, pointing to the large gap between inherent parallelism in the programs and what commercial compilers of the time were able to exploit. The Polaris Group performed such a comparison for 13 programs from the Perfect benchmark suite [4,8]. They cited the need for compilers to incorporate array privatization and interprocedural analysis, among other things, to exploit a coarser granularity of parallelism. These early studies focused developers of commercial and research compilers to investigate incorporating these techniques, and now they are beginning to make their way into practice. As stated earlier, our experiment goes beyond these previous studies because it measures parallelism potential empirically using run-time testing. Further, now that these previously missing techniques are performed automatically by the SUIF compiler, a new experiment can identify the next set of missing analysis techniques.

Analysis techniques exploiting predicates have been developed for specific data-flow problems including constant propagation [31], type analysis [28], symbolic analysis for parallelization [5,12], and the array data-flow analysis described above [11,29]. Tu and Padua present a limited sparse approach on a gated SSA graph that is demand based, only examining predicates if they might assist in loop bounds or subscript values for parallelization, a technique that appears to be no more powerful than that of Gu, Li and Lee [29]. Related to these array analysis techniques are approaches to enhance scalar symbolic analysis for parallelization. Haghighat describes an algebra on control flow predicates [12] while Blume presents a method for combining control flow predicates with ranges of scalar variables [5]. As compared to these previous approaches [5,11,12,28,29,31] our approach is distinguished in several ways: (1) it is capable of deriving low-cost run-time tests, consisting of arbitrary program expressions, to guard conditionally optimized code; (2) it incorporates predicates other than just control flow tests, particularly those derived from the data-flow values using predicate extraction; and (3) it unifies a number of previous approaches in array data-flow analysis, as previously discussed.

Some previous work in run-time parallelization uses specialized techniques not based on data-flow analysis. An inspector/executor technique inspects array accesses at run time immediately prior to execution of the loop [23,24]. The inspector decides whether to execute a parallel or sequential version of the loop. Predicated data-flow analysis instead derives run-time tests based on values of scalar variables that can be tested prior to loop execution. Thus, our approach, when applicable, leads to much more efficient tests than inspecting all of the array accesses within the loop body.

There are some similarities between our approach and much earlier work on data-flow analysis frameworks. Holley and Rosen describe a construction of qualified data-flow problems, but with only a fixed, finite, disjoint set of predicates [17]. Cousot and Cousot describe a theoretical construction of a reduced cardinal power of two data-flow frameworks, in which a data-flow analysis is performed on the lattice of functions between the two original data-flow lattices, and this technique has been refined by Nielson [7,21]. Neither of the latter two prior works were designed with predicates as one of the data-flow analysis frameworks, and none of the three techniques derives run-time tests.

Recently, additional approaches that, in some way, exploit control-flow information in data-flow analysis have been proposed [2,6,25]. Ammons and Larus's approach improves the precision of data-flow analysis along frequently taken control flow paths, called hot paths, by using profile information. Bodík et al. describe a demand-driven interprocedural correlation analysis that eliminates some branches by path specialization. Both approaches utilize code duplication to sharpen data-flow values but are only applicable if the information is available at compile time. Deferred data-flow analysis proposed by Sharma et al. attempts to partially perform data-flow analysis at run time, using control-flow information derived during execution.

7. Conclusion and future work

This paper has presented the results of an experiment to determine whether there are remaining opportunities for improving automatic parallelization systems for a set of 17 programs from two benchmark suites. With a model where a loop's iterations can only execute in parallel if it accesses independent memory locations, possibly after privatization of array data structures, we have identified all the remaining loops not parallelized by the SUIF compiler for which parallelization is safe. Our results indicate that there is still some room for improvement in automatic parallelization in two areas: incorporating control flow tests into analysis and extracting low-cost run-time tests wherever applicable instead of using an inspector/executor. These two requirements can be met with a single new analysis technique, predicated array data-flow analysis, whereby predicates are associated with data-flow values. We have shown preliminary results that predicated array data-flow analysis can improve the speedup for three out of the four programs in the SPECFP95 benchmark suite that previously did not speed up well.

In future work, we envision extending predicated data-flow analysis to derive more aggressive run-time tests to enable parallelization of additional loops. A particularly interesting area of future study is how to integrate run-time tests derived from predicated array data-flow analysis with an inspector/executor approach. We observed a few inherently parallel loops in our experiments where predicated array data-flow analysis is not applicable, such as when arrays are used in subscript expressions. The run-time tests arising from predicated analysis and an inspector/executor approach are complementary, and a parallelization system that combines the two techniques and uses array data-flow analysis to optimize run-time tests as much as possible is desirable to exploit all of the remaining inherent parallelism in these programs.

References

[1] S.P. Amarasinghe, Parallelizing compiler techniques based on linear inequalities, PhD thesis, Dept. of Electrical Engineering, Stanford University, January 1997.

[2] G. Ammons and J.R. Larus, Improving data-flow analysis with path profiles, in: *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998, pp. 72–84.

[3] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger and P. Tu, Parallel programming with Polaris, *IEEE Computer* **29**(12) (December 1996), 78–82.

[4] W. Blume and R. Eigenmann, Performance analysis of parallelizing compilers on the Perfect Benchmark programs, *IEEE Trans. Parallel Distrib. Systems* **3**(6) (November 1992), 643–656.

[5] W.J. Blume, Symbolic analysis techniques for effective automatic parallelization, PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, June 1995.

[6] R. Bodík, R. Gupta and M.L. Soffa, Interprocedural conditional branch elimination, in: *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 1997, pp. 146–158.

[7] P. Cousot and R. Cousot, Systematic design of program anaysis frameworks, in: *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, San Antonio, TX, January 1979, pp. 269–282.

[8] R. Eigenmann, J. Hoeflinger and D. Padua, On the automatic parallelization of the Perfect Benchmarks, *IEEE Trans. Parallel Distrib. Systems* **9**(1) (January 1998), 5–23.

[9] G. Goff, Practical techniques to augment dependence analysis in the presence of symbolic terms, Technical Report TR92–194, Dept. of Computer Science, Rice University, October 1992.

[10] J. Gu, Z. Li and G. Lee, Symbolic array dataflow analysis for array privatization and program parallelization, in: *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[11] J. Gu, Z. Li and G. Lee, Experience with efficient array data-flow analysis for array privatization, in: *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, Las Vegas, NV, June 1997, pp. 157–167.

[12] M.R. Haghighat, Symbolic analysis for parallelizing compilers, PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, August 1994.

[13] M.W. Hall, S.P. Amarasinghe, J.M. Anderson, B.R. Murphy, S.-W. Liao and M.S. Lam, Interprocedural parallelization analysis in SUIF, *ACM Trans. Programming Languages Systems* (1999), to appear.

[14] M.W. Hall, S.P. Amarasinghe, B.R. Murphy, S.-W. Liao and M.S. Lam, Detecting coarse-grain parallelism using an interprocedural parallelizing compiler, in: *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[15] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.-W. Liao, E. Bugnion and M.S. Lam, Maximizing multiprocessor performance with the SUIF compiler, *IEEE Computer* **29**(12) (Dec. 1996), 84–89.

[16] M.W. Hall, B.R. Murphy, S.P. Amarasinghe, S.-W. Liao and M.S. Lam, Interprocedural analysis for parallelization, in: *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995, pp. 61–80.

[17] L. Howard Holley and B.K. Rosen, Qualified data flow problems, in: *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, Las Vegas, NV, January 1980, pp. 68–82.

[18] F. Irigoin, Interprocedural analyses for programming environments, in: *Proceedings of the NSF-CNRS Workshop on Environment and Tools for Parallel Scientific Programming*, Sept. 1992.

[19] S. Moon and M.W. Hall, Evaluation of predicated array data-flow analysis for automatic parallelization, in: *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, Atlanta, GA, May 1999, pp. 84–95.

[20] S. Moon, M.W. Hall and B.R. Murphy, Predicated array data-flow analysis for run-time parallelization, in: *Proceedings of the 1998 ACM International Conference on Supercomputing*, Melbourne, Australia, July 1998, pp. 204–211.

[21] F. Nielson, Expected forms of data flow analysis, in: *Programs as Data Objects*, H. Ganzinger and N.D. Jones, eds, Lecture Notes on Comput. Sci., Vol. 217, Springer, Berlin, 1986, pp. 172–191.

[22] W. Pugh and D. Wonnacott, Eliminating false data dependences using the Omega test, in: *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992, pp. 140–151.

[23] L. Rauchwerger and D. Padua, The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization, in: *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995, pp. 218–232.

[24] J.H. Saltz, R. Mirchandaney and K. Crowley, Run-time parallelization and scheduling of loops, *IEEE Trans. Computers* **40**(5) (May 1991), 603–612.

[25] S.D. Sharma, A. Acharya and J. Saltz, Defered data-flow analysis: Algorithms, proofs and applications, Technical Report UMD-CS-TR-3845, Dept. of Computer Science, University of Maryland, November 1997.

[26] J.P. Singh and J.L. Hennessy, An empirical investigation of the effectiveness and limitations of automatic parallelization, in: *Proceedings of the International Symposium on Shared Memory Multiprocessing*, April 1991.

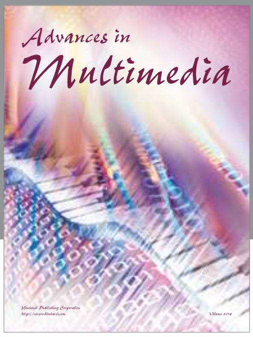
[27] B. So, S. Moon and M.W. Hall, Measuring the effectiveness of automatic parallelization in SUIF, in: *Proceedings of the 1998 ACM International Conference on Supercomputing*, Melbourne, Australia, July 1998, pp. 212–219.

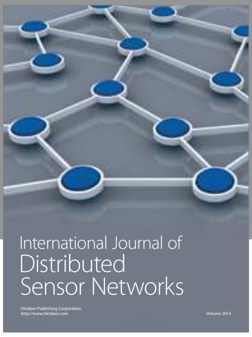
[28] R.E. Strom and D.M. Yellin, Extending typestate checking using conditional liveness analysis, *IEEE Trans. Software Eng.* **19**(5) (May 1993), 478–485.

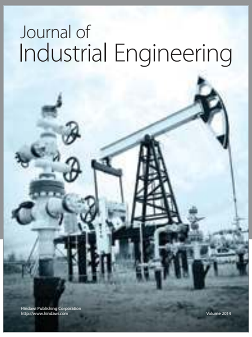
[29] P. Tu, Automatic array privatization and demand-driven symbolic analysis, PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, May 1995.

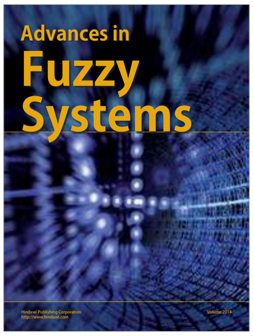
[30] P. Tu and D. Padua, Automatic array privatization, in: *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, Aug. 1993, pp. 500–521.

[31] M.N. Wegman and F. Kenneth Zadeck, Constant propagation with conditional branches, *ACM Trans. Programming Languages Systems* **13**(2) (April 1991), 180–210.









Hindawi

Submit your manuscripts at
http://www.hindawi.com