

# Run-time Parallelization Optimization Techniques <sup>\*</sup>

Hao Yu and Lawrence Rauchwerger <sup>\*\*</sup>

Department of Computer Science  
Texas A&M University  
College Station, TX 77843-3112  
{h0y8494,rwenger}@cs.tamu.edu

**Abstract.** In this paper we first present several compiler techniques to reduce the overhead of run-time parallelization. We show how to use static control flow information to reduce the number of memory references that need to be traced at run-time. Then we introduce several methods designed specifically for the parallelization of sparse applications. We detail some heuristics on how to speculate on the type and data structures used by the original code and thus reduce the memory requirements for tracing the sparse access patterns without performing any additional work. Optimization techniques for the sparse reduction parallelization and speculative loop distribution conclude the paper.

## 1 Run-Time Parallelization Requires Compiler Analysis

Current parallelizing compilers cannot identify a significant fraction of parallelizable loops because they have complex or statically insufficiently defined access patterns. To fill this gap we advocate a novel framework for their identification: speculatively execute the loop as a `doall`, and apply a fully parallel data dependence test to determine if it had any cross-processor dependences; if the test fails, then the loop is re-executed serially. While this method is inherently scalable its practical success depends on the fraction of ideal speedup that can be obtained on modest to moderately large parallel machines. Maximizing the resulting parallelism can be obtained only through a minimization of the run-time overhead of the method, which in turn depends on its level of integration within a restructuring compiler. This technique (the LRPD test) and related issues have been presented in detail in [3, 4] and thus will not be presented here.

We describe a compiler technique that reduces the number of memory references that have to be collected at run-time by using static control flow information. With this technique we can remove the shadowing of many references

---

<sup>\*</sup> A full version of this paper is available as Technical Report TR99-025, Dept. of Computer Science, Texas A&M University

<sup>\*\*</sup> Research supported in part by NSF CAREER Award CCR-9734471, NSF Grant ACI-9872126, DOE ASCI ASAP Level 2 Grant B347886 and a Hewlett-Packard Equipment Grant

that are redundant for the purpose of testing valid parallelization. Moreover we group 'similar' or 'related' references and represent them with a single shadow element, thus reducing the memory and post-execution analysis overhead of our run-time techniques. We introduce a method that speculates on the actual data structures and access patterns of the original sparse code and compacts the dynamically collected information into a much smaller space. We further sketch an adaptive optimization method for parallelizing reductions in irregular and sparse codes. Other parallelism enabling loop transformations, e.g., loop distribution, requiring precise data dependence analysis are applied speculatively and tested at run-time without additional overhead.

The presented techniques have been implemented in the Polaris compiler [1] and employed in the automatic parallelization of some representative cases of irregular codes: SPICE 2G6 and P3M.

## 2 Redundant Marking Elimination

**Same-Address Type Based Aggregation.** While in previous implementations we have traced every reference to the arrays under test we have found that such an approach incorporates significant redundancy. We only need to detect attributes of the reference pattern that will insure correct parallelization of loops. For this purpose memory references can be classified, similar to [2] as: Read only (RO), Write-first (WF), Read-first-write (RW) and Not referenced (NO). NO or RO references can never introduce data dependences. WF references can always be privatized. RW accesses must occur in only one iteration (or processor) otherwise they will cause flow-dependences and invalidate the speculative parallelization. The overall goal of the algorithm is to mark only the necessary and sufficient sites to unambiguously establish the type of reference: WF,RO,RW or NO by using the dominance (on the control graph) relationship.

Based on the control flow graph of the loop we can aggregate the marking of read and/or write references (**to the same address**) into one of the categories listed above and replace them with a single marking instruction. The algorithm relies on a DFS traversal of the control dependence graph (CDG) and the recursive combination of the elementary constructs (elementary CDG's). The final output of the algorithm is a loop with fewer marks than the number of memory references under test. If predicates of references are loop invariant then the access pattern can be fully analyzed before loop execution in an inspector phase. This inspector can be equivalent to a LRPD test (or simpler run-time check) of a generalized address descriptor.

**Grouping of Related References.** Two memory addresses are **related** if they can be expressed as a function of the same base pointer. For example, when subscripts are of the form  $ptr + \text{affine function}$ , then all addresses starting at the pointer  $ptr$  are related. Intuitively, two related references of the same type can be aggregated for the purpose of marking if they are executed under the same control flow conditions, or more aggressively, if the predicates guarding of one reference imply the other reference.

A *marking group* is a set of subscript expressions of references to an array under test that satisfies the following conditions: (a) the addresses are derived from the same base pointer, (b) for every path from the entry of the considered block to its exit all *related* array references are of the same type, i.e., have the same attribute (WF, RO, RW, NO). The *grouping algorithm* tries to find a minimum number of disjoint sets of references of maximum cardinality (subscript expressions) to the array under test. Once these groups are found, they can be marked as a single abstract reference. The net result is a reduced number of marking instructions (because we mark several individual references at once) and a reduced size (dimension) of the shadow structure that needs to be allocated because several distinct references are mapped into a single marking point. Similarly, a **Global Reference Aggregation** can be achieved when groups of references formed with different base pointers occur under the same conditions.

### 3 Some Specific Techniques for Sparse Codes

The essential difficulty in sparse codes is that the dimension of the array tested may be orders of magnitude larger than the number of distinct elements referenced by the parallelized loop. Therefore the use of shadow arrays will cause the allocation of too much memory and generate useless work during the phases of the test making it not scalable with data size and/or number of processors.

**Shadow Structures for Sparse Codes.** Many sparse codes use linked structure traversals when processing their data. The referenced pointers can take any value (in the address space) and give the overall 'impression' of being very sparse and random. We use a compile time heuristic to determine the type of data structure and access pattern employed by the program and then, speculatively, use a conformable shadow data structure. Correct speculation results in minimal overhead, otherwise the parallelization becomes more expensive.

We have identified several situations where such a technique is beneficial, the most notable one being the use of linked lists in a loop. We classify the accesses of the base-pointers used by such a loop as (a) monotonic with constant stride, (b) monotonic with variable stride and (c) random. For each of these possible reference patterns we have adopted a specialized representation: (i) monotonic constant strides are recorded as a triplet [offset, stride, count], (ii) monotonic references with variable stride are recorded in an array and a tuple for their overall range [min, max], (iii) random addresses use hash tables (for large number of references) or simple lists to be sorted later and a tuple for their range.

The run-time marking routines are adaptive: They will verify the class of the access pattern and use the simplest possible form of representation. Ideally all references can be stored as a triplet, dramatically reducing the space requirements. In the worst case, the shadow structures will be proportional to the number of marked references. The reference **type** (WF, RO, RW, NO) will be recorded in a **bit vector** possibly as long as the number recorded references.

After loop execution the analysis of the recorded references will detect collisions using increasingly more complex algorithms: (1) Check for overlap of

address ranges traversed by the base pointers (linked lists) using range information. (2) If there is overlap then check (analytically) triplets for collisions; Check collision of monotonic stride lists by merging them into one array (3) Sort random accesses stored in lists (if they exist) and merge into other the previous arrays. (4) Merge hash tables (if they exist) into the previous arrays.

### 3.1 Sparse Reduction Parallelization through Selective Privatization

Usually reductions are parallelized by accumulating in private arrays conformable to their shared counterpart and then, after loop execution, merged in parallel on their shared data. Such an approach is not beneficial if reductions are sparse because the final update (merge) would require much more work than necessary, since only a fraction of the private arrays is actually modified during the loop execution. Moreover, if the contention to the reduction elements is low, then privatization through *replication* across *all* processors is also sub-optimal.

We developed a hybrid method that first compacts the reduction references on every processor through the use of private conformable (across processors) hash tables and then uses the collected cross-processor collision information to selectively privatize only those reduction elements that cause contention across processors. Furthermore, this information can be reused during subsequent instantiations of the loop without the need to hash the reference pattern.

### 3.2 Speculative Loop Distribution

Loops with statically unavailable access patterns cannot be safely transformed because the required data dependence analysis is not possible. We adopt a speculative approach in which we assume that certain pointers are not aliased and prove the transformation correct at run-time. For example, we apply the distribution of a linked list traversal out of a loop assuming that it is not modified by the remainder of the loop. This condition is then verified at run-time together with the rest parallelization conditions (which subsume it) without additional overhead. This method has been applied in the parallelization of loops in SPICE.

## References

1. W. Blume *et. al.* Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.
2. J. Hoeflinger. *Interprocedural Parallelization Using Memory Classification Analysis*. PhD thesis, University of Illinois, August, 1998.
3. L. Rauchwerger. Run-time parallelization: A framework for parallel computation. TR. UIUCDCS-R-95-1926, Dept of Comp. Science, University of Illinois, Sept. 1995.
4. L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Trans. on Parallel and Distributed Systems*, 10(2), 1999.
5. J. Wu, *et. al.* Runtime compilation methods for multicomputers. In Dr. H.D. Schwetman, editor, *Proc. of the 1991 Int. Conf. on Parallel Processing*, pages 26–30. CRC Press, Inc., 1991. Vol. II - Software.