Extending the Applicability and Improving the Performance of Runtime Parallelization

Shun-Tak Leung and John Zahorjan

Department of Computer Science and Engineering
University of Washington

Technical Report 95-01-08
January 1995

Extending the Applicability and Improving the Performance of Runtime Parallelization

Shun-Tak Leung and John Zahorjan *
Department of Computer Science & Engineering
University of Washington

January 1995

Abstract

When static analysis of a sequential loop fails to yield reliable information on its dependence structure, a parallelizing compiler is left with three alternatives: it can take the conservative option of emitting code for a sequential execution; it can optimistically emit code to speculatively execute the loop as a DOALL [6, 7]; or it can emit inspector-executor code to determine the actual dependence structure at runtime and to respect it in a parallel execution [8, 9]. The first approach is certain to yield a slow execution. The second approach gives very good results when the loop can in fact be executed as a DOALL, but is of no help otherwise.

In this paper we concentrate on the final approach, runtime parallelization through the inspector-executor method. We have two goals in this work. The first is to expand the class of loop to which the approach may be applied by removing restrictions on the loop dependence structures that it can handle. To achieve this goal, we introduce new forms of the inspector and executor that together remove all restrictions on the loop dependence structure. Thus, we show how to parallelize a class of loop that previously would have compelled the compiler to emit sequential code.

Our second goal is to improve the performance of the inspector-executor approach through specializations applicable when static analysis yields some (weak) information about the array indexing functions used in assignments. We validate our work through a set of examples designed to illustrate the fundamental performance tradeoffs characterizing the specialized implementations, using results taken from executions on 32 processors of a KSR1.

1 Introduction

To parallelize a sequential loop at compile time, a compiler must compute a parallel schedule of the iterations based on a static analysis of loop-carried dependences. Some loops, however, may contain parallelism not detectable in this way. For example, in sparse matrix computations, array subscripts often involve indirection arrays and thus defy static analysis [6, 9].

There are two basic approaches that can be taken to parallelizing such loops: speculative execution as a DOALL [6, 7]; and the inspector-executor method [8, 9]. While the former yields good results when the

*This material is based upon work supported by the National Science Foundation (Grants CCR-9123308 and CCR-9200832), the Washington Technology Center, and Digital Equipment Corporation Systems Research Center and External Research Program. Authors' addresses: Department of Computer Science & Engineering, University of Washington, Seattle, WA 98195; *shuntak@cs.washington.edu, zahorjan@cs.washington.edu.*

loop is in fact executable as a DOALL, it will fail in other cases, and sequential execution of the loop will be required.

In this paper we examine the second approach, the inspector-executor method. Like much of the previous work in this area [3, 9], we focus on implementations for shared address space multiprocessors.

```
do i = 1, n
  x[i] = F(x[g(i)], ...)
enddo
```

(a) Source Loop: Restricted Form

```
do i = 1, n
  x[f(i)] = F(x[g(i)], ...)
enddo
```

(b) Source Loop: General Form

Figure 1: Sequential Source Loop

Let us call the sequential loop to be parallelized the *source loop*. Figure 1(a) (adapted from [8]) shows the form of the source loops addressed in previous work [8, 9]. (Since we are concerned only about loop-carried dependences, statements not causing such dependences are omitted. Similarly, we have shown a singly nested loop for simplicity only; the techniques are applicable to more general loop nesting structures.) That work assumes that the indexing function used in assignment is the identity function, i.e., the assignment made in iteration i is to element i of the array. This restriction means that the loop has no loop-carried output dependences, and that $x[i]$ is written by iteration i (only). These two observations make possible the inspector and executor algorithms in [9], shown here in Figure 2. To parallelize the source loop, the compiler generates two pieces of code: an *inspector* and an *executor*. At run time, the inspector examines loop-carried dependences and computes a parallel schedule. The executor then performs the loop iterations, using the parallel schedule provided by the inspector.

```
/* wf is initially all zero */
do i = 1, n
  wf[i] = max(wf[g(i)],...) + 1
enddo
```

(a) Basic Inspector

```
do w = 1, W
  doall i such that wf[i] = w
    if (g(i)<i) t1 = x2[g(i)] else t1 = x1[g(i)]
    ...
    x2[i] = F(t1, t2, ...)
  enddo
enddo

x1 = x2
```

(b) Basic Executor

Figure 2: Current Inspector and Executor for Runtime Parallelization

In Figure 2(a), the inspector computes a wavefront schedule respecting flow dependences only. While output dependences in the source loop are prohibited, antidependences are not. Thus, antidependences must be handled by the executor (Figure 2(b)). The key to doing this is using two arrays for x. Notice that because output dependences are prohibited, the source loop writes each element at most once. Before an element is written, it contains an "old" value; afterwards, a "new" value. Arrays $x1$ and $x2$ store the old and new values respectively. The executor writes computed values only into $x2$. When it reads an element, it accesses either $x1$ or $x2$ depending on whether the sequential loop would read the old or new value. After each loop execution, $x2$ is copied to $x1$ to prepare for another execution.

Figure 1(b) shows a more general form of the source loop. Here the indexing function used in assignment is not the identity function, but instead is a general function, $f(i)$. Because nothing is known statically about the form of $f(i)$, we cannot assume that there are no output dependences in the loop, as is required by the current technology. Finding an inspector-executor method for this more general source loop structure is one

goal of our work. To do this, we propose a new form for the inspector-executor, in which the inspector takes into account all forms of dependence and the executor merely executes iterations according to the inspector's schedule.

A second goal of our work is to investigate ways of improving performance of the inspector-executor method. We do this by specializing the implementations based on weak information about the array indexing functions that may be available to the compiler statically. Additionally, because we have introduced a new form for the inspector-executor method, we also examine the fundamental performance trade-offs between it and the traditional inspector-executor approach.

Section 2 describes the two approaches in more detail. Section 3 discusses their performance implications qualitatively. Section 4 reports measurement results. Section 5 concludes this paper.

2 Extending the Applicability and Improving Performance

We begin this section by presenting a new inspector-executor formulation that allows general dependences of the form shown in Figure 1(b). We refer to the traditional approach as approach A, and our new method as approach B. In subsequent subsections, we show how to improve the performance of the type B inspector-executors by specializing them to take advantage of restricted forms of the indexing function $f(i)$. We also show how to extend the type A approach for these specializations, borrowing some ideas from the type B approach. The final result is a single approach (type B) for the most general form of the loop, and type A and type B approaches for the more restricted forms. The tradeoffs between the two approaches (when both apply) are examined in the remainder of this paper.

2.1 Approach B: Inspector Handles All Dependences

Figure 3 shows the type B inspector and executor for the most general source loop (Figure 1(b)). Compared with its type A counterpart in Figure 2(a), the type B inspector uses two new arrays: lr, to detect antidependences, and lw, to detect flow and output dependences.

$lr[i]$ records the last wavefront that reads $x[i]$. Suppose there is an antidependence from iterations k to l, which means that iteration k reads $x[g(k)]$, iteration l writes $x[f(l)]$, $g(k) = f(l)$, and $k < l$. If the inspector assigns iteration k to, say, wavefront w, it updates $lr[g(k)]$ such that $lr[g(k)] \geq w$. When it later comes to iteration l, it chooses a wavefront that is, among other things, after wavefront $lr[f(l)]$ (i.e., $lr[g(k)]$). This ensures that iteration l is executed after wavefront w and hence iteration k, thus respecting the antidependence. A similar explanation applies to lw.

Because it handles more general loops, the type B inspector is clearly more complicated than the type A inspector. On the other hand, the type B executor is simpler, since in the type B scheme the executor is not concerned with enforcing any dependences. Moreover, the type B approach assumes nothing about $f(i)$ and can therefore be used for any $f(i)$.

2.2 Specializing for Performance

In this subsection we consider opportunities to simplify the type B approach (and so improve its performance), and to slightly extend the type A method. We do this by specializing each approach based on the information available about the indexing function $f(i)$, which controls which element of array x is written by iteration i.

```
/* wf,lr,lw are initially all zero */
do i = 1,n
  w = max(lr[f(i)],lw[f(i)],lw[g(i)],...) + 1
  lw[f(i)] = wf[i] = w
  lr[g(i)] = max(lr[g(i)],w)
  ...
enddo
```

(a) Type B Inspector: General $f(i)$

```
do w = 1,W
  doall i such that wf[i] = w
    x[f(i)] = F(x[g(i)],...)
  enddo
enddo
```

(b) Type B Executor

Figure 3: Type B Approach

We consider four cases: $f(i)$ is the identity function, $f(i)$ is strictly monotonic, $f(i)$ is a general invertible function, and $f(i)$ is a general function.

Case 1: Identity Function $f(i) = i$

This is the base case for the type A approach. Its inspector and executor are shown in Figure 2.

Figure 4(a) shows the type B inspector specialized for this case. In comparison with the general type B inspector, the array lw has been replaced with the array wf of the original (type A) inspector: because $f(i)$ is the identify function, it is known *a priori* that element i is written by iteration i (only), and so determining the time of the last write to element i is equivalent to determining the wavefront to which iteration i is scheduled. For a similar reason, checking $lw[f(i)]$ is unnecessary.

```
/* wf,lr are initially all zero */
do i = 1,n
  wf[i] = max(lr[i],wf[g(i)],...) + 1
  lr[g(i)] = max(lr[g(i)],wf[i])
  ...
enddo
```

(a) Type B Inspector: $f(i) = i$

```
/* wf,lr,lw are initially all zero */
do i = 1,n
  w = max(lr[f(i)],lw[g(i)],...) + 1
  lw[f(i)] = wf[i] = w
  lr[g(i)] = max(lr[g(i)],w)
  ...
enddo
```

(b) Type B Inspector: Invertible $f(i)$

Figure 4: Specialized Type B Inspectors

Case 2: Strictly Monotonic $f(i)$

We assume that $f(i)$ is statically known to be strictly monotonic increasing, a common example being an affine function with a positive coefficient (i.e., $f(i) = ki + c$ where $k > 0$). The case for decreasing is analogous and therefore omitted. Figure 5(a) shows the type A executor .

Compared with the earlier algorithm (Figure 2(a)), the type A inspector needs an additional array lw. In the earlier algorithm, $wf[i]$ carries two pieces of information: iteration i is executed in wavefront $wf[i]$; $x[i]$ is written in wavefront $wf[i]$. They are synonymous if $f(i) = i$, but this no longer holds. Thus, in the present algorithm (Figure 5(a)), $lw[i]$ stores the second piece of information — the wavefront that writes $x[i]$.

As for the executor (Figure 5(b)), since $f(i)$ is strictly monotonic, the loop writes each element at most once and thus we can use the existing technique of storing old and new values in two arrays. When the executor reads $x[k]$ in iteration i, it must decide which array to access depending on whether the sequential

```
/* wf, lw initially all zero */
do i = 1, n
  wf[i] = lw[f(i)] = max(lw[g(i)],...) + 1
enddo
```

(a) Type A Specialized Inspector

```
do w = 1, W
  doall i such that wf[i] = w
    k = g(i)
    if (k<f(i)) then t1 = x2[k] else t1 = x1[k]
    ...
    x2[f(i)] = F(t1, t2, ...)
  enddo
enddo

x1 = x2
```

(b) Type A Executor: Strictly Monotonic Increasing $f(i)$

```
do w = 1, W
  doall i such that wf[i] = w
    k = g(i)
    if (f_inv[k]<i) then t1 = x2[k] else t1 = x1[k]
    ...
    x2[f(i)] = F(t1, t2, ...)
  enddo
enddo

x1 = x2
```

(c) Type A Executor: General Invertible $f(i)$

Figure 5: Specialized Type A Inspectors and Executors

loop would write $x[k]$ before or after iteration i. Suppose iteration j writes $x[k]$ (i.e., $k = f(j)$). The executor needs to know only whether $i < j$, not j itself. Since $f(i)$ is strictly monotonic increasing, $i < j$ if and only if $f(i) < f(j)$. Thus, the executor simply compares $f(i)$ and k, without finding j. An important point is that, compared with the existing executor algorithm (Figure 2(b)), there is no new overhead as the array subscripts must be computed and a comparison done anyway.

In a type A executor, assuming $f(i)$ to be strictly monotonic simplifies the decision on which array to access for read operands. As the type B executor needs not make this decision, the assumption has no effect and so does not result in a new, specialized form of the inspector and executor.

Case 3: Invertible $f(i)$

Figure 5(a) and (c) show the type A inspector and executor in this case. As $f(i)$ is invertible, each array element is written at most once. Therefore, the type A inspector needs not be changed and, as before, we can store old and new values of x in two arrays. However, to decide which array to read, the type A executor needs an explicit mapping (namely $f^{-1}(i)$) indicating which iteration writes a given element. Array f_inv stores this mapping.

Looking up f_inv is a non-trivial executor overhead because it involves an extra memory operation per read reference. Also, computing f_inv is a preprocessing overhead in addition to the inspector itself. Both overheads can be avoided if $f^{-1}(i)$ can be statically and symbolically determined, but this seems unlikely except for affine $f(i)$ (which would be strictly monotonic and thus fall under case 2).

When $f(i)$ is statically determined to be invertible, we can use the type B inspector in Figure 4(b). It differs from the fully general type B inspector in that it needs not check the time of the last write of element $f(i)$ (i.e., $lw[f(i)]$) when it places iteration i in a wavefront: since the indexing function $f(i)$ is invertible, the current write to element $f(i)$ must be the only write.

Case 4: Unrestricted $f(i)$

An unrestricted $f(i)$ may cause multiple writes to the same element. It is unclear how to handle this with the type A approach. Conceivably, there could be multiple arrays for multiple versions of x. The executor would need other data structures to decide which version each iteration should access to read the correct

value, that is, the one that the same iteration would read in the sequential execution. The overheads of storage management, deciding which version to read, and keeping track of and looking up the additional data structures are likely to be much larger than those we have seen so far.

The case of unrestricted $f(i)$ is the base case for the type B approach (Figure 3), which has no difficulty in dealing with it.

3 Comparing Performance of the Two Approaches Qualitatively

In the previous section we introduced a new inspector-executor scheme with the primary goal of extending the class of loops that could be handled. We also discovered specializations of both the existing (type A) approach and the new (type B) approach for cases in which both are applicable. In this section, we contrast the performance implications of the two approaches in terms of several issues. First, we examine the potential advantage of the type B executor over the type A, based on the latter's need to execute code enforcing antidependences. Next, we discuss the impact of respecting antidependences in the inspector's schedule on schedule depth, and so on performance.

3.1 Executor Overheads

As discussed earlier, under approach A we store data in two arrays. The executor (Figure 2(b)) writes results into one, and copies it to the other after loop execution. For each element read, the executor must decide which array should be accessed. By contrast, under approach B, the executor (Figure 3(b)) needs none of these, and therefore generally has less memory and runtime overheads.

Although under approach A we must allocate memory for the extra array, we can often avoid or reduce copying. Copying is unnecessary if the loop writes every element it reads. Successive executor invocations can alternate the roles of the two arrays: one invocation reads old values from $x1$ and writes new results to $x2$; the next reads $x2$ and writes $x1$. If the compiler cannot ascertain whether the loop writes every element it reads (because, for example, array subscripts are not known statically), copying can be reduced by copying the entire $x1$ to $x2$ once and for all before the first executor invocation, and alternating usage of the two arrays in subsequent invocations. After the initial copying, corresponding elements in $x1$ and $x2$ that the loop does not write will always contain identical values. The executor can access either array for these values.

Under approach A, the overhead of deciding which array to access grows linearly with the number of elements read by the loop. How significant it is depends on how the decision is made. In the simple cases (see Figure 2(b) for $f(i) = i$ and Figure 5(b) for strictly monotonic $f(i)$), it is just a comparison with no extra memory operations; the comparison's operands are needed for array indexing anyway. However, in more general cases (e.g., Figure 5(c) for invertible $f(i)$), extra information has to be fetched from data structures in memory, which can be a significant overhead.

3.2 Effect on Parallel Schedule

Under approach A, the schedule respects only flow dependences; for B, it respects antidependences as well. For any given source loop, the schedule under approach B therefore has at least as many wavefronts as that under approach A. Deeper schedules do not necessarily increase execution times proportionally, but do

increase the total synchronization overhead (because there are more wavefronts) and the potential for load imbalance within wavefronts (because each has, on the average, fewer iterations).

However, in many common cases, schedules for the two approaches are identical. Obviously, they are identical if the loop contains only flow dependences, as in sparse lower-triangular solve. A less obvious case is when an antidependence between two iterations always implies a flow dependence. In this case, a schedule respecting the flow dependences automatically respects the antidependences as well. An example is using successive overrelaxation (SOR) [1, 5] to solve a sparse linear system with a symmetric coefficient matrix[1]. Such linear systems may arise from finite elements analysis in structural engineering or numerical solution of elliptic partial differential equations using finite difference [5].

In addition to depth, the schedules computed under the two approaches also affect executor cache behaviors differently in ways that are highly loop-specific. Because memory access times are increasingly important determinants of overall performance, this effect can have a pronounced influence on the relative costs of approaches A and B. However, because the reference pattern is highly sensitive to the specific problem, the effect is data dependent, and in our experience does not favor either approach with regularity.

4 Experimental Measurements

We now present measurement results illustrating the performance tradeoffs discussed earlier. In doing so, we are of course limited to cases in to both approaches A and B apply. It is important to remember, however, that a primary motivation for the development of approach B was that it extends the class of loop to which runtime parallelization can be applied beyond what is possible following approach A.

Our source loop is one iteration of the successive overrelaxation (SOR) algorithm for solving a sparse linear system [1]. ([6] list a number of applications taken from the Perfect Benchmarks that defy static analysis, and to which the inspector-executor approach is applicable.) The dependences in the SOR application depend on the sparsity structure of the coefficient matrix, thus making compile-time parallelization impossible. Because our goal is to understand the influence of the fundamental distinctions between the type A and type B approaches, rather than to report their running times on a specific problem, we used synthetic coefficient matrices in our examples. By doing so, we were able to isolate and control problem characteristics that might influence the comparison of the two approaches, such as the depth of the parallel schedules resulting from each approach and the total amount of work to be done by the executors. Figure 6 shows relevant characteristics of the matrices we used.

The experiments were run on a Kendall Square Research KSR1 shared-memory multiprocessor [2] running OSF/1. All programs were written in C using KSR1's *pthreads* and the source loop was manually transformed into different inspectors and executors. We employed a runtime restructuring technique we have developed [4] to reduce the adverse cache effects of runtime parallelization under both approaches. Timings are for 32 processors on one ring.

For approach A, we compared the array copying costs with iteration execution times. Also, we compared inspector and executor performance under approaches A and B for two cases: $f(i) = i$ and $f(i)$ given by an indirection array statically known to be invertible[2]. For approach B, the executor in Figure 3(b) was used

[1] In this loop, each iteration computes a component of the solution vector (x). Given the coefficient matrix B, iteration k reads $x[l]$ (written by iteration l) if and only if $b_{kl} \neq 0$. As B is symmetric, $b_{kl} \neq 0 \Leftrightarrow b_{lk} \neq 0$, and hence iteration l reads $x[k]$ if and only if iteration k reads $x[l]$. Each antidependence implies a corresponding flow dependence (and, in this example, vice versa).

[2] Indirection arrays often contain input-dependent data that defy compiler-time analysis. For our purpose of performance comparison between approaches A and B, however, we assume compile-time knowledge of an invertible $f(i)$ so that there is a type A executor that can be used.

Name	Order	# Nonzeros	Schedule Depth	
			Type A	Type B
M1	100000	1145000	20	20
M2	100000	1147500	20	40
M3	100000	1148750	20	80
M4	100000	1149375	20	160
N1	40000	458000	20	20
N2	40000	459000	20	40
N3	40000	459500	20	80
N4	40000	459750	20	160
O1	100000	290000	20	20
O2	100000	575000	20	20
O3	100000	765000	20	20
O4	100000	955000	20	20

Figure 6: Characteristics of Sparse Coefficient Matrices

in both cases. For approach A, the base executor (Figure 2(b)) was used in the first case and the executor for invertible $f(i)$ (Figure 5(c)) in the second.

4.1 Array Copying

Figure 7 shows the array copying costs and iteration execution times (for $f(i) = i$) under approach A. As expected, copying cost are roughly the same for all matrices in series O, whereas iteration execution times increase with the number of nonzeros. Also, the copying costs are small relative to iteration execution times. Noting that copying can often be avoided or reduced (see Section 3.1), we conclude that array copying is not a main concern in the tradeoff between approaches A and B.

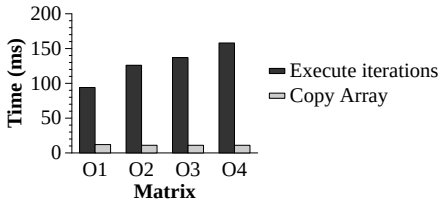


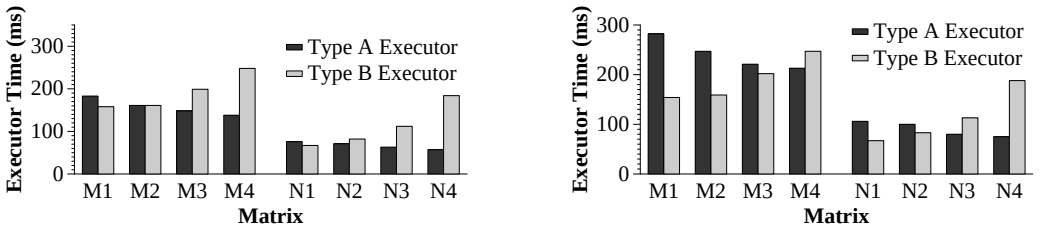
Figure 7: Costs of Array Copying vs. Iteration Execution

4.2 Cost of Executing Iterations

We now look at iteration execution times[3] for both approaches in two cases: $f(i) = i$ (executors shown in Figure 2(b) and Figure 3(b)) and $f(i)$ given by an indirection array (executors shown in Figure 5(c) and Figure 3(b)). Figure 8 shows the results.

Consider the execution times for $f(i) = i$ (Figure 8(a)). We first focus on matrices M1 and N1, for which the schedules under approaches A and B have the same number of wavefronts. (In fact, they happen to be identical.) Therefore, the two executors suffer roughly the same synchronization overhead and load imbalance. The type B executor runs 10–15% faster than the type A executor because the former needs not

[3] In our experiments for approach A, the array *was* copied, although the timings exclude copying. Omitting the copying would affect our timings by creating a data access pattern different from that of a "real" type A executor, which does the copying.



(a) $f(i) = i$ (b) $f(i)$ is an indirection array

Figure 8: Type A vs. Type B Executors

decide which array to access for each element read. The difference in data access pattern[4] may also have played a role, but its effect is not entirely clear.

Next, we look at the trend within each matrix series in Figure 8(a). As the difference in the depths of the two schedules increases, the performance advantage of the type B executor decreases and is finally negated[5]. The executor suffers larger synchronization overhead between wavefronts and potentially greater load imbalance within wavefronts because there are more wavefronts and, on the average, fewer iterations per wavefront. If, however, there are many iterations per wavefront, both effects will diminish because the absolute synchronization cost would be a small percentage of the per-wavefront compute time, and having more iterations available for scheduling simplifies load balancing. This is evident from comparing the results for series M and N.

Figure 8(b) shows the execution times of the executor for invertible $f(i)$ (Figure 5(c) and Figure 3(b)), where $f(i)$ is given by an indirection array. For M1 and N1, the type B executor is about 40% faster than the type A executor. This difference is larger than for $f(i) = i$ because, for each element read, the type A executor must look up an array f_inv (see Figure 5(c)), while the type B executor does not. This extra memory read has poor locality since its reference pattern is determined by the coefficient matrix's irregular sparsity structure. As the depth of type B schedules increases (within each matrix series), however, the performance advantage of the type B executor is again gradually offset by synchronization costs and load imbalance.

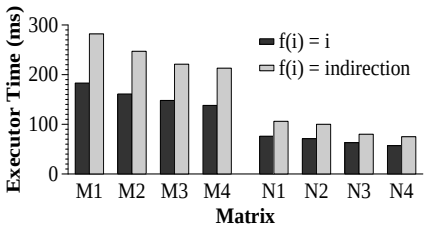
Finally, we assess the importance of identifying optimization opportunities. Figure 9 compares the execution times of the simpler executor for $f(i) = i$, and the more complicated but general executor for invertible $f(i)$, where $f(i)$ is given by an indirection array. The indirection represented an identity mapping. If this were known statically, the compiler would have generated the first, simpler executor. Executor performance for approach A is sensitive to whether the compiler can exploit such opportunities. In our experiments, the second, overly general executor takes 30% to 50% longer than the simpler one (see Figure 9(a)). As for approach B (Figure 9(b)), one form of executor handles all kinds of loop uniformly.

4.3 Inspector Performance and Overall Execution Time

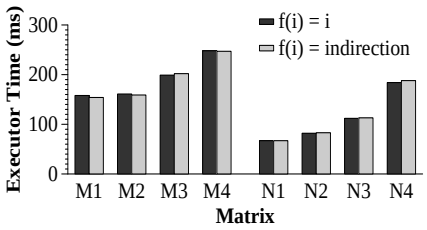
Figure 10 shows the inspector execution times. (All inspectors ran sequentially.) As expected, the type B inspector, being more complicated, takes substantially longer (roughly 70% in most cases we measured).

[4] Although the schedules are identical, the data access patterns are different because the type A executor reads some values from one array and some from the other, whereas the type B executor always reads the same array.

[5] Although the matrices in each series represent nominally the same amount of work, they lead to different access patterns and thus execution times. We are interested only in how the *difference* between approaches A and B for each matrix varies.



(a) Type A Executor



(b) Type B Executor

Figure 9: Importance of Identifying Optimization Opportunities

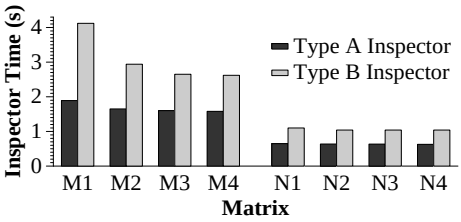


Figure 10: Inspector Performance: A vs. B

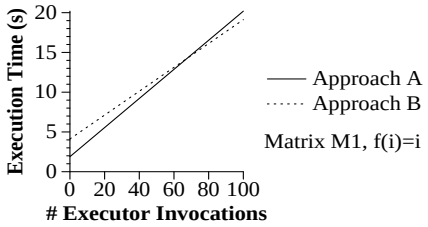


Figure 11: Overall Execution Time: A vs. B

Often the source loop is executed many times with the same dependence pattern [8, 9]. In these cases, the inspector is executed once and each executor invocation reuses the same schedule. Figure 11 illustrates how the overall execution time depends on the number of executor invocations. In this example, the type B executor is faster than the corresponding type A executor. If the executor is executed many times, approach B will have a shorter overall execution time than approach A. The exact cross-over point is problem-specific. Sometimes the type B executor is slower than the corresponding type A executor, in which case approach B will always take longer overall.

5 Conclusion

In this paper, we have extended the applicability of runtime parallelization by generalizing the class of loops that can be handled efficiently. We examined two approaches to handling general dependences. In approach A, the inspector computes a schedule that respects flow dependences but leaves the handling of anti- and output dependences to the executor. In approach B, the schedule is guaranteed to respect all dependences; the executor needs only follow the schedule.

For each approach, we found optimizations that can be applied to special circumstances. We compared their performance qualitatively and reported measurements on a KSR1. Overall, approach B is more general and flexible, while approach A often has better performance, at least for simple dependence patterns. A compiler wishing to use runtime parallelization for loops that defy a sufficiently accurate static dependence analysis should probably employ both approaches, choosing between them depending on the characteristics of the loop.

References

[1] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice Hall, Englewood Cliffs, 1989.

[2] Henry III Burkhardt, Steven Frank, Bruce Knobe, and James Rothnie. Overview of the KSR1 computer system. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, February 1992.

[3] Shun-Tak Leung and John Zahorjan. Improving the performance of runtime parallelization. In *Proceedings of Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–91, May 1993.

[4] Shun-Tak Leung and John Zahorjan. Restructuring arrays for efficient parallel loop execution. Technical Report 94-02-01, Department of Computer Science and Engineering, University of Washington, 1994.

[5] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical recipes: the art of scientific computing.* Cambridge University Press, Cambridge, 1986.

[6] Lawrence Rauchwerger and David Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. Technical Report CSRD-TR-1390, Center for Supercomputing Research and Development, University of Illinois, Urbana-Champaign, IL.

[7] Lawrence Rauchwerger and David Padua. The privatizing DOALL test: A run-time technique for DOALL loop identification and array privatization. In *Proceedings of International Conference on Supercomputing*, 1994.

[8] Joel Saltz, Harry Berryman, and Janet Wu. Multiprocessors and runtime compilation. In *Proceedings of International Workshop on Compilers for Parallel Computers, Paris*, 1990.

[9] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Runtime parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.