Combining Flow and Dependence Analyses to Expose Redundant Array Accesses

Elana Granston
Alexander Veidenbaum

CRPC-TR95545
May 1995

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

To appear in the *International Journal of Parallel Programming*

Combining Flow and Dependence Analyses to Expose Redundant Array Accesses

Elana D. Granston
Center for Research on Parallel Computation
Rice University
6100 S. Main Street, Houston, Texas 77005
e-mail: granston@cs.rice.edu

Alexander V. Veidenbaum
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana, Illinois 61801
e-mail: sasha@csrd.uiuc.edu

Abstract

The success of large-scale hierarchical and distributed shared memory systems hinges on our ability to reduce delays resulting from remote accesses to shared data. To facilitate this, we present a compile-time algorithm for analyzing programs with doall-style parallelism to determine when read and write accesses to shared data are *redundant* (unnecessary). Once identified, redundant remote accesses can be replaced by local accesses or eliminated entirely. This optimization improves program performance in two ways. First, slow memory accesses are replaced by faster ones. Second, the time to perform other remote memory accesses may be reduced as a result of the decreased traffic level. We also show how the information obtained through redundancy analysis can be used for other compiler optimizations such as prefetching and cache management.

Key words: redundancy detection, communication analysis, flow analysis of parallel programs, dependence analysis, and data reuse.

1 Introduction

Large-scale, high-performance, hierarchical and distributed shared memory systems are currently gaining in popularity. Although research prototypes of such systems have existed for some time [GKLS83, LP92, LLJ+92, KDCZ94], recently commercial versions such as the Kendall Square Research KSR1 and KSR2 [Ken92], the Cray T3D [Cra93], and the CONVEX Exemplar [CON93] have appeared on the market, and plans to build others have been announced. Although there is significant architectural variation between these systems, all of them have some local storage facilities — for example, cache and software-controllable local memory — that are cheaper to access than remote memory.

The success of these systems depends heavily on the compiler's ability to reduce delays due to remote memory accesses. One method for accomplishing this is by eliminating *redundant* read and write accesses to remote copies of shared data. Informally, suppose that a processor performs a read of shared data. The read is *redundant* if

there was an earlier read and write of those data by that same processor and the data that were read or written at that earlier point would still be valid at this later read. A write of shared data is redundant if the only processor that will read those data before they become stale is the processor that performed the write.

If the compiler can detect redundant references, then it can perform compile-time optimizations that make use of local storage facilities to either replace redundant remote read and write accesses by local accesses or else eliminate the redundant accesses entirely. Eliminating a redundant access can speed up program execution in two ways. First, a fast local access replaces a slow remote access. Second, the elimination of remote accesses decreases the overall traffic level, and hence the amount of contention, which in turn allows the remaining remote accesses to complete more quickly.

The compiler can detect and eliminate redundant references to scalar data in sequential programs relatively easily with the use of well-established techniques for eliminating redundant computations [MR79, RWZ88]. However, handling parallel numerical applications adds additional levels of complexity. First, redundant accesses to *array* data become increasingly important to detect.[1] Second, the partitioning of the computation across processors must be considered.

In this paper, we discuss the notion of *redundancy analysis* with respect to data accesses in programs that have already been manually or automatically parallelized. We also detail a complete compile-time algorithm for detecting unconditionally redundant reads in such programs and discuss adaptations of this algorithm to provide other redundancy-related information that would be useful for compiler optimizations such as prefetching and cache management.

The remainder of this paper is organized as follows. Section 2 describes the parallelism model that we assume and provides examples that show how information obtained from redundancy analysis can be used to reduce delays arising from non-local memory accesses. Section 3 formally defines and categorizes redundancies. Section 4 presents the basic forward flow analysis algorithm for detecting unconditionally redundant reads. Section 5 covers adaptations of the algorithm presented in this paper for computing other types of redundancy information and for handling other parallel program models. Section 6 addresses related work. Section 7 presents conclusions.

2 Motivation

The algorithm described in this paper computes information regarding both redundant references and those references that *induce* or cause these redundancies. In the case of parallel programs, the the compiler's ability to detect redundancies depends on the availability of information regarding the scheduling of loop iterations both within and across loops. In Section 2.1, we present and justify the scheduling strategy assumed in this paper. In Section 2.2, we demonstrate the importance of computing redundancy information by presenting examples to show how the results of such analysis could be used to reduce delays due to shared data accesses on today's systems.

[1] Although Rosen et al. [RWZ88] also handle the case of individual array elements when subscript expressions are equivalent, handling just this limited case is insufficient when trying to aggressively optimize parallel applications.

Figure 1: Static inter-epoch scheduling. Original program. (left) Same program after static inter-epoch scheduling. (right)

2.1 Parallel Program Model

We assume a global name-space and a single level of doall-style parallelism. Data is designated as *private* or *shared*. The doall constructs partition the program into *parallel epochs* (doall loops) and *sequential epochs*. In a parallel epoch, no datum written during one iteration is accessed during another, so all iterations of the doall loop can be executed in parallel. Each doall loop iteration and sequential epoch constitutes a separate epoch *instance*.

We can schedule epoch instances using either static or dynamic *inter-epoch* scheduling. In a dynamically scheduled program, each epoch boundary (doall or enddoall statement) corresponds to an actual processor reassignment boundary. Each doall corresponds to a fork, and each enddoall to a join. We make no assumptions at compile time regarding the assignment of epoch instances to processors after crossing a processor reassignment boundary.

A program that utilizes static inter-epoch scheduling, executes one fork at the beginning of the program and one join at the end. Other epoch boundaries correspond to barriers. Note that static inter-epoch scheduling is different from classical static scheduling. Classical static scheduling refers to the compile time partitioning of iterations within a single doall loop. Static inter-epoch scheduling refers to compile time partitioning of iterations across multiple doall loops. Both types of static scheduling can be combined, for example, by stripmining the doall loops as a preprocessing step. Further note that static inter-epoch scheduling is essentially SPMD-style scheduling[2] which is becoming increasingly popular over the fork-join model (dynamic inter-epoch scheduling). This popularity is due in part to the lower overhead of SPMD-style scheduling in the presence of short epoch instances, which are common in practice, and in part to the additional opportunities to exploit locality that can be exposed at compile time [AL93, AHD93, BGM95].

With static inter-epoch scheduling, the compiler must choose some technique for mapping iterations to processors. Although there are several methods for assigning loop iterations to processors, advocating specific scheduling policies is not our goal. Although our algorithm can be applied using any number of strategies, for presentation purposes, we adopt one simple strategy that we expect will work reasonably well. With this strategy, each sequential epoch is enclosed by a conditional that ensures that it is only executed by virtual processor p_{seq}. If

[2]Despite the popularity of the term *SPMD*, we retain the term *static inter-epoch scheduling* both for compatibility with our previous work and because we believe that this term more clearly describes the key scheduling assumptions that are used in this paper.

```
shared A,B,C
private x

doall i=1 to N
   A[i] = ···
   B[i] = ···
enddoall


doall i=1 to N
   x = A[i-1] + A[i]
   do j=1 to N
      B[i] = (x + B[i]) / 2 + C[j,i]
   enddo
enddoall


doall i=1 to N
   if ···
      then
         do j=1 to N
            ··· = B[i] + C[j,i]
         enddo
         B[i] = 0
      else
         B[i] = - B[i]
   endif
enddoall
```

```
shared A,B,C
private x,tA,tB

doall i=1 to N
   tA[i] = ···
   A[i] = tA[i]
   tB[i] = ···
enddoall

doall i=1 to N
   x = A[i-1] + tA[i]
   do j=1 to N
      tC[j,i] = C[j,i]
      tB[i] = (x + tB[i]) / 2 + tC[j,i]
   enddo
enddoall

doall i=1 to N
   if ···
      then
         do j=1 to N
            ··· = tB[i] + tC[j,i]
         enddo
         B[i] = 0
      else
         B[i] = - tB[i]
   endif
enddoall
```

Figure 2: The reads of `A[i]` and `B[i]` in the second doall loop and the reads of `B[i]` and `C[j,i]` in the third doall loop are unconditionally redundant, as are the writes of `B[i]` in the the first and second doall loops. The inducers of these redundancies are the writes of `A[i]` and `B[i]` in the first doall loop, the write of `B[i]` in the second doall loop, and the read of `C[j,i]` in the second doall loop. These redundant references can potentially cause non-local communication because of false sharing effects, cache conflicts, and/or the use of write-through caches. (left) The redundant reads and writes of shared variables are eliminated by using private copies. (right)

iteration i of one doall loop is executed on virtual processor p_i, then iteration i of every doall loop is executed on p_i. For example, in Figure 1 (right), static scheduling is applied to the set of doall loops from Figure 1 (left). Without loss of generality, we will henceforth assume that virtual processor $p_i = i$ and that $p_{seq} = p_1$.

2.2 Exploiting Redundancy Information

Suppose that local storage facilities consist of private (per-processor) caches. The results of redundancy analysis can be used to improve cache utilization both on systems with and without hardware support for coherence. On the Cray Research T3D, for example, only private data is cached by default. The compiler can override this default for sets of shared data by using directives or making private copies of data to be cached, but then it assumes the responsibility of maintaining coherence for these data. Compiler-directed coherence algorithms [Vei86, CKM88, CV88, DMCK92] must be conservative. Therefore, the number of unnecessary invalidations that the compiler inserts is inversely proportional to the precision of the information available at compile time.

Because the results of redundancy analysis provide more detailed information on stale data (equivalently, non-redundant data) than was previously available [CV88], the compiler can be more sparing regarding the number of invalidation instructions that it inserts. For example, consider Figure 2 (left). Because of our scheduling assumptions, the references to `B[i]` are redundant. Therefore, a cache copy of `B[i]` is definitely valid on entrance to the third doall loop and should be used, if available. However, a cache coherence scheme that does not consider both array subscripts and compile-time knowledge regarding the scheduling of loop iterations would have to conservatively assume that a cache copy of `B[i]` might be stale. Therefore, the compiler would have to (unnecessarily) insert an invalidate instruction for `B` between the second and third doall loops.

In a system such as the Kendall Square KSR1, cache coherence is supported in hardware using an invalidate protocol. In such a system, the hardware might conservatively invalidate copies of cache lines that contain valid data that will still be reused. As an example of this, refer to the second doall loop in Figure 2 (left) where array `B` may be falsely shared. Moreover, the ping-pong effect resulting from this false sharing can be severe, causing unnecessary non-local reads and writes as well as non-local communication needed to perform the invalidations.

The false sharing problem can be avoided if each processor makes a local (private) copy of the data it needs and reads this copy instead.[3] In general, the compiler should create a private copy or update one whenever a write of a shared variable is determined to induce a redundancy. Whenever a read of shared variables is determined to be redundant, the compiler should substitute this shared read with a read of the private copy. These two optimizations will expose redundant writes of shared variables that the compiler can then eliminate.

The compiler can also apply the technique of making private copies to optimize accesses to `A`. In this case, data copying makes a local copy of the value of `A[i]` that is written in the first doall loop available within the second doall loop. The compiler can use redundancy information to determine when to create, update, and read these private copies. Referring to Figure 2 (left), for example, the writes of `A[i]` and `B[i]` and the read of `C[j,i]` in the first two doall loops induce redundancies, so the compiler should create or update local copies at these points. Then the compiler can replace the redundant reads of `A[i]` and `B[i]` in the second doall loop and the the redundant reads of `B[i]` and `C[j,i]` in the third doall loop by reads of private copies. Afterwards, the compiler can eliminate the redundant writes of `B[i]` in the first and second doall loops and that of `A[i]` in the second doall loop. The transformed code is shown in Figure 2 (right). Note that, although the write to `A[i]` induces a redundancy, it is not in itself redundant. Therefore, the compiler cannot eliminate the write to the shared copy of this variable. Further note that to automatically detect that the latter write to `B[i]` in the second loop is redundant requires a compile-time algorithm that can handle both conditional code and doall loops.

In all the above cases, the compiler's ability to replace reads and writes to shared data by reads and writes to private copies depended on the information that the data being written would not be needed by another processor. This knowledge can be supplied by redundancy analysis. The ability to make private copies can improve performance in other situations as well. First, memory access delays take varying amounts of time depending on the overall traffic level. On systems such as the T3D which employ a write-through cache, applying the aforementioned technique of making a private copy of `B[i]` would eliminate $N(N+1)$ of the $N(N+2)$ writes to `B` in this example, thereby reducing the system traffic level. Second, on systems such as the T3D, where caches are small and associativity is low, the compiler can apply data copying to significantly reduce the number

[3] To reduce the size of private variables, loops should be stripmined based on cache capacity whenever possible. For simplicity and increased readability, we omit stripmining from the examples in this paper.

of shared data misses due to cache conflicts [TGJ93]. Although we did not consider the use of registers in the examples presented above, the private copies could just as easily have been register-allocated, assuming that a sufficient number of registers were available [BJEW91].

Therefore, basic redundancy analysis is useful both for optimizing cache coherence protocols and for determining when private copies can be safely made. As will be discussed briefly in Section 5.4, some minor extensions would allow this analysis to be used for determining the range in which data can be safely prefetched, so that latencies resulting from non-redundant reads can be hidden.

Observe that, even for the simple example presented in this section, the analysis technique must consider array subscript expressions and control flow that includes conditionals, loops, and doall constructs. Therefore, a combined flow and dependence analysis approach is needed.

3 Definitions

Before we can present our redundancy detection algorithm, we need more precise definitions of the terms redundancy and redundancy inducer. In practice, not all redundancies can be profitably eliminated. Therefore, we also classify redundancies from most promising to least promising, so that we can restrict ourselves to detecting those that can be profitably eliminated on the target architecture. Section 3.1 develops formal definitions for sequential programs and Section 3.2 extends these definitions to parallel programs.

3.1 Redundancies in Sequential Programs

Suppose that reference R is a read. If the data accessed at R are read or written earlier in the program, then R is redundant. Our algorithm detects read redundancies in two steps. First, it computes the set of data that are accessed prior to executing R, known as the set of data that *downwardly reach* R. Then, the algorithm intersects this reaching set with the set of data read at R to determine whether R is redundant. The redundancy inducers are the earlier references from which the data in the intersection set reach.

Suppose instead that R is a write reference. If the data written at R are written later in the program, then R is redundant. Our algorithm for detecting write redundancies also has two steps. First, it computes the set of data that are written after executing R, known as the set of data that *upwardly reach* R. Then, the algorithm intersects this reaching set with the set of data written at R to determine whether R is redundant. The redundancy inducers are the later references from which the data in the intersection set reach.

Sections 3.1.1 through 3.1.3 formally define the concepts of reaching, redundancy, and redundancy inducer for sequential programs.

3.1.1 Reaching Data in Sequential Programs

Unless specified otherwise, let a *reference* R to an array variable X refer to a static read or write of one or more x, where x is an element of a shared array X. Without loss of generality, each program statement is assumed to contain at most one shared reference. Let P be a sequential program with one entry point and one exit point, s be a statement in P, and ρ be an execution path from the start of P to s, excluding s. If there exists a memory reference R along ρ that accesses a variable x, and the value of x after completing the access at R is the same as that immediately preceding execution of s, then x *downwardly reaches* s *from* R *along* ρ.

```
     shared A

     if ···
        then
             do i=1 to N by 2
R0 :            A[i] = ···
             enddo
     endif
     if ···
        then
             do i=1 to N by 3
R1 :            ··· = A[i]
             enddo
     endif
```

Figure 3: Some elements of array `A` that are read at R_1 might be written earlier at R_0.

The term *upwardly reaching* can be defined analogously. Let ρ be an execution path from s to the end of P, excluding s. If there exists a memory reference R along ρ that accesses a variable x, and the value of x immediately after executing s is the same as that immediately preceding the access at R, then x *upwardly reaches* s.

The concept of *downwardly reaching* varies subtly from the classical definition of *reaching* [ASU86], where data reach from definitions (writes) only. For our purposes, data can reach from either a read or write reference, since either can produce a locally available data copy. Furthermore, our goal is to optimize accesses to shared data, so only references to shared data are considered during the computation of reaching data. We also extend the concept of reaching so that data can reach both downward (forward) and upward (backward).

Reaching data can be further partitioned into *conditionally* or *unconditionally* reaching data. If there exists at least one path ρ along which x *might* be referenced, then x *conditionally* reaches s. If x is definitely referenced along every path from the start (end) of P to s, then the stronger statement can be made that x *unconditionally* downwardly (upwardly) reaches s. For example, in Figure 3, `A[1:N:2]` are accessed along some path from the beginning of the program to R_1. Therefore, `A[1:N:2]` conditionally downwardly reach reference R_1 from R_0. Meanwhile, in Figure 4, `B[1:N:2]` are written along *all* execution paths from R_0 to the end of the program. Therefore, `B[1:N:2]` unconditionally upwardly reach reference R_0 from the pair of references R_1 and R_2.

3.1.2 Redundancies in Sequential Programs

Intuitively, a static reference R is *redundant* if an access to the shared copy at this point is at least partially unnecessary or could be made so by reusing locally available data. Either an earlier read or write can cause a read reference to be redundant. In contrast, only a later write can induce another write reference to be redundant. In other words, read redundancies can arise from either flow or input dependences, but write redundancies can only arise from output dependencies.

Suppose R references at least one element x of array X. A read reference R is

- *fully redundant* if every x referenced at R unconditionally downwardly reaches R from one or more read

```
     shared B

     do i=1 to N by 2
R0 :    B[i] = ···
     enddo
     if ···
        then
             do i=1 to N
R1 :            B[i] = ···
             enddo
        else
             do i=1 to N by 2
R2 :            B[i] = ···
             enddo
     endif
```

Figure 4: Every element of array `B` written at R_0 is also written at both R_1 and R_2.

references, write references, or a combination thereof;

- *partially redundant*[4] if there exists an x referenced at R that unconditionally downwardly reaches R from one or more read references, write references, or a combination thereof; or
- *conditionally redundant* if there exists an x referenced at R that conditionally downwardly reaches R from some read or write reference.

A write reference R is

- *fully redundant* if every x referenced at R unconditionally upwardly reaches R from one or more write references;
- *partially redundant* if there exists an x referenced at R that unconditionally upwardly reaches R from one or more write references; or
- *conditionally redundant* if there exists an x referenced at R that conditionally upwardly reaches R from some write reference.

In the case of both read and write redundancies, the first category of redundancies, namely full redundancies, is retained from classical redundancy detection algorithms. The second category, namely partial redundancies, arises only when dealing with array region accesses. The third includes all potential redundancies. By definition, every full redundancy is also partially redundant; every partial redundancy is also conditionally redundant. Full and partial redundancies are collectively referred to as *unconditional redundancies*, because at least one datum must be redundant. Any references involving unresolvable aliases should not be considered as potential redundancies or redundancy inducers.

For example, in Figure 3, `A[1:N:3]` are written at R_1. A subset of these data, namely `A[1:N:6]`, conditionally downwardly reaches R_1 from R_0. Therefore, R_1 is conditionally read redundant. In Figure 4, recall that array

[4] The term *partially redundant* is used differently here than in Morel and Renvoise [MR79].

elements `B[1:N:2]` unconditionally upwardly reach R_0 from the pair of references R_1 and R_2, because `B[1:N:2]` are written along both branches of the if statement. Therefore, R_0 is fully (unconditionally) write redundant.

3.1.3 Redundancy Inducers in Sequential Programs

As mentioned earlier, references that cause other references to become redundant are known as redundancy inducers. Recall that read redundancies are the sinks of flow and input dependences. Therefore, read redundancy inducers are the sources of flow and input dependences. Meanwhile, write redundancies are the sources of output dependences, so write redundancy inducers are the sinks of output dependences. Consequently, a read redundancy can be induced by earlier reads and/or writes, but a write redundancy can only be induced by later writes.

More formally, a read or write reference R

- *fully* induces read redundancies if every x referenced at R unconditionally upwardly reaches R from one or more read references;
- *partially* induces read redundancies if there exists an x referenced at R that unconditionally upwardly reaches R from one or more read references; or
- *conditionally* induces read redundancies if there exists an x referenced at R that conditionally upwardly reaches R from some read reference.

A write reference R

- *fully* induces write redundancies if every x referenced at R unconditionally downwardly reaches R from one or more write references;
- *partially* induces write redundancies if there exists an x referenced at R that unconditionally downwardly reaches R from one or more write references; or
- *conditionally* induces write redundancies if there exists an x referenced at R that conditionally downwardly reaches R from some write reference.

By definition, every fully redundancy-inducing reference is also partially redundancy-inducing; every partially redundancy-inducing reference is also conditionally redundancy-inducing. Fully and partially redundancy-inducing references are jointly classified as *unconditionally* redundancy-inducing references.

Examples of redundancy inducers can also be seen in Figures 3 and 4. In Figure 3, the data written at R_1 conditionally upwardly reach R_0, so R_0 conditionally induces a read redundancy. In Figure 4, the data written at R_0 unconditionally downwardly reach R_1 and R_2. Only some of the data written at R_1 are written at R_0, so R_1 partially (unconditionally) induces a write redundancy at R_0. In contrast, all data written at R_2 are written at R_0, so R_2 fully (unconditionally) induces a write redundancy at R_0.

3.2 Redundancies in Parallel Programs

In the context of a parallel program, when is a reference R redundant? Suppose that R is a read reference and that the reference to R by processor p (more precisely, the instance of R executed by processor p) causes a read of array element `A[i]`. Then R is redundant if and only if

```
        shared A

        doall i=1 to N
R0:         A[i] = ···
R1:         ··· = A[i]
        enddoall

        doall i=1 to N
R2:         ··· = A[i-1]
        enddoall
R3:     ··· = A[N]

        doall i=1 to N
R4:         ··· = A[i]
R5:         A[i] = ···
        enddoall
```

Figure 5: R_0 induces a read redundancy at R_1. R_0 and R_1 induce a read redundancy at R_4. R_2 and R_3 prevent R_5 from inducing a write redundancy at R_0.

- there is an earlier read or write reference to `A[i]` by processor p *and*
- there is *no* intervening write of `A[i]` by processor $p' \neq p$ in between the earlier reference and R.

These conditions jointly ensure that `A[i]` is read or written earlier by processor p and that the shared copy of `A[i]` is not modified by another processor in between the potential save and reuse points.

If R is redundant, then there exists at least one reference (possibly more) that induces the redundancy. Let R' be a read or write reference to `A[i]` by processor p, and assume that R' is executed before R. Suppose that there is no intervening write by *any* processor (including p) between R' and R. Then the same value is referenced at R' and R, so R' induces a redundancy at R.

Suppose instead that R is a write reference. R is redundant if and only if

- there is a later write reference to `A[i]` by *any* processor *and*
- there is *no* intervening read of `A[i]` by processor $p' \neq p$ in between R and the later write of `A[i]`.

These conditions jointly ensure that the value of `A[i]` that processor p writes will never be read by any another processor.

At least one reference must induce the write redundancy at R. Let R' be a write of `A[i]` by any processor that is executed after R. Suppose that there is no intervening write of `A[i]` between references R and R' by *any* processor. Then the value of `A[i]` that is written at R is overwritten at R', so R' induces the redundancy.

As an example, consider Figures 5 and 6. Assume that static inter-epoch scheduling is used, as described in Section 2.1. (A discussion of dynamic inter-epoch scheduling is postponed until Section 5.5.) Let $p = p_i$. In Figure 5, the value of `A` written at R_0 by processor p is read by p at R_1 and again at R_4. Therefore, reference R_0 induces a redundancy at R_1. Both R_0 and R_1 induce the read redundancy at R_4. The value written at R_0 by processor p is also read at R_2 and R_3 by processors other than p. Therefore, R_0 is *not* write redundant. In

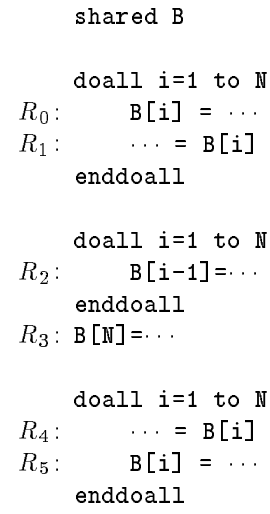


Figure 6: R_0 induces a read redundancy at R_1. R_2 and R_3 prevent R_0 and R_1 from inducing a read redundancy at R_4. R_2 and R_3 induce a write redundancy at R_0. R_4 prevents R_5 from inducing write redundancies at R_2 and R_3.

Figure 6, `B[i]` is written at R_0, read at R_1 and then read at R_4. However, in this case, the value of `B` written at R_0 by processor p is overwritten at R_2 and R_3 before being read by any processors other than p. Therefore, R_0 is write redundant. R_2 and R_3 induce this redundancy. R_2 and R_3 also prevent R_4 from being redundant. Meanwhile, R_4 prevents R_5 from inducing write redundancies at R_2 and R_3.

Note that, when dealing with parallel programs, the case for write redundancies is *not* symmetric to the case for read redundancies. Consider the case of write redundancies. The presence of an intervening read can affect whether a particular write reference is redundant, but the presence of an intervening write cannot. Meanwhile, the presence of an intervening write can affect the set of writes that induce a particular write redundancy. In contrast, in the case of read redundancies, the presence of an intervening write can prevent a read redundancy, whereas the presence of an intervening read has no effect. Another difference between the case of write and read redundancies is that a redundant write can be induced by a write of similar data by any processor. A redundant read can only be induced by a read of similar data by the same processor.

A reference R' can only induce another reference R to be redundant if the data referenced at R' reaches R. To extend the definition of *reaching* to parallel programs, the definition must be constrained so that data cannot reach across redundancy-preventing intervening references. In the case of write redundancy detection, the definition must allow data referenced by one processor to reach a reference instance executed by another processor. Unlike the sequential case, the criteria for a datum to reach a read reference now differs from that for reaching a write reference.

4 Detecting Redundancies and Their Inducers

Recall that redundancy identification proceeds in two steps. First, flow analysis is used to compute the data sets that reach each basic block and the references from which they reach. Then, these sets are used to identify redundancies and their inducers. Section 4.1 discusses data set representation. Section 4.2 describes the basic algorithm for detecting read redundancies in sequential programs. Section 4.3 extends this algorithm to handle parallel programs, and Section 4.4 addresses complexity issues. Adaptations for identifying write redundancies and redundancy inducers are postponed until Section 5.

4.1 Data Structures and Operations

Although the sets computed during flow analysis are described as data sets, they are implemented as sets of *data descriptors*. A data descriptor D contains the following fields:

$\mathcal{VAR}(D)$	—	the variable name,
$\mathcal{SHAPE}(D)$	—	a list of summary shape(s) describing the region(s) of $\mathcal{VAR}(D)$ being referenced, and
$\mathcal{REF}(D)$	—	a list of reference sets. Each reference set in $\mathcal{REF}(D)$ corresponds to exactly one summary shape in $\mathcal{SHAPE}(D)$.

There are four binary operations on data descriptors or sets thereof: union ($\bigcup$), intersection ($\bigcap$), difference ($-$), and reference difference ($-_R$). The first three are intuitive. The fourth operator $-_R$ eliminates references associated with a particular data set without changing the data set itself. Therefore, $-_R$ is similar to $-$, with one exception: $-$ modifies both the $\mathcal{SHAPE}$ and the $\mathcal{REF}$ fields of the left-hand operand, while $-_R$ modifies only the $\mathcal{REF}$ field of the left-hand operand. An example of applying these operators is presented in Figure 7.

As will be seen later, the sets used exclusively as intermediaries to generate array kill information during flow analysis do not need reference information. Therefore, in the data descriptors of these sets, only $\mathcal{VAR}$ and $\mathcal{SHAPE}$ fields are needed. Using only these two fields saves space and simplifies the resulting flow analysis equations.

The method for representing the summary shapes must meet the following requirements. First, the representation must maintain information about the external shape (boundary) and internal shape (contiguity) of the represented regions. Second, it must be possible to compute the intersection, union, and difference of two summary shapes. Third, it must be possible to estimate the results of the above operations in either direction. Last, the shape must be represented as a function of the indices of enclosing loops, so that the reference instance corresponding to a particular iteration can be represented.

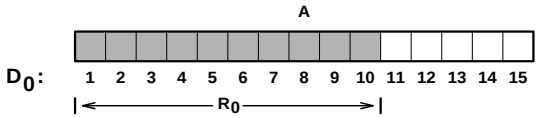
It is straightforward to adapt existing shape representation methods [Bal90, HK91, Sch89] and exact dependence analysis techniques such as [HHL90, Pug92] to meet these requirements. Hence, one of the strengths of the algorithm that will be presented in Section 4.2 is that it is not dependent on any particular method for representing array regions and computing set operations. Depending on the complexity of the selected methods and number of summary shapes used to represent each region, accuracy-space-time trade-offs can be made.

4.2 Algorithm for Detecting Redundancies in Sequential Programs

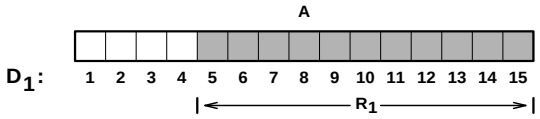
This section presents our flow-analysis algorithm for detecting unconditional redundancies in programs targeted for uniprocessors. Section 4.3 covers extensions to handle parallel programs. Although iterative flow-analysis

Suppose that `A[1:10]` are referenced at R_0 and `A[5:15]` are referenced at R_1. Let D_0 and D_1 be data descriptors that describe the regions being referenced at R_0 and R_1, respectively. Then,

- $\mathcal{VAR}(D_0) =$ `A`, $\mathcal{SHAPE}(D_0) = \{\{1:10\}\}$, and $\mathcal{REF}(D_0) = \{\{R_0\}\}$.



- $\mathcal{VAR}(D_1) =$ `A`, $\mathcal{SHAPE}(D_1) = \{\{5:15\}\}$, and $\mathcal{REF}(D_1) = \{\{R_1\}\}$.

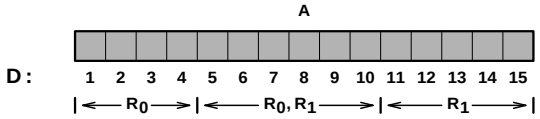


and the result of

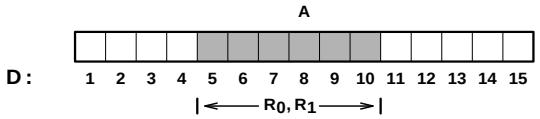
$$D = D_0 \ op \ D_1$$

is as follows:

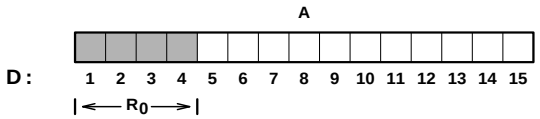
- If $op = \bigcup$, then $\mathcal{VAR}(D) =$ `A`, $\mathcal{SHAPE}(D) = \{\{1:4\},\{5:10\},\{11:15\}\}$ and $\mathcal{REF}(D) = \{\{R_0\},\{R_0,R_1\},\{R_1\}\}$.



- If $op = \bigcap$, then $\mathcal{VAR}(D) =$ `A`, $\mathcal{SHAPE}(D) = \{\{5:10\}\}$ and $\mathcal{REF}(D) = \{\{R_0,R_1\}\}$.



- If $op = -$, then $\mathcal{VAR}(D) =$ `A`, $\mathcal{SHAPE}(D) = \{\{1:4\}\}$ and $\mathcal{REF}(D) = \{\{R_0\}\}$.



- If $op = -_R$, then $\mathcal{VAR}(D) =$ `A`, $\mathcal{SHAPE}(D) = \{\{1:4\},\{5:10\}\}$ and $\mathcal{REF}(D) = \{\{R_0\},\phi\}$.

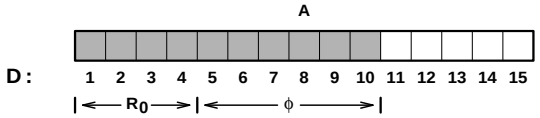


Figure 7: Example of applying set operators.

UREF(b): Set of data that unconditionally downwardly reach the end of block b from references within b, and the references from which these data reach.

UIN(b): Set of data that unconditionally downwardly reach the start of block b, and the references from which these data reach.

UOUT(b): Set of data that unconditionally downwardly reach the end of block b, and the references from which these data reach.

CREF^w(b): Set of data that conditionally downwardly reach the end of block b from write references within b.

CIN^w(b): Set of data that conditionally downwardly reach the start of block b from write references.

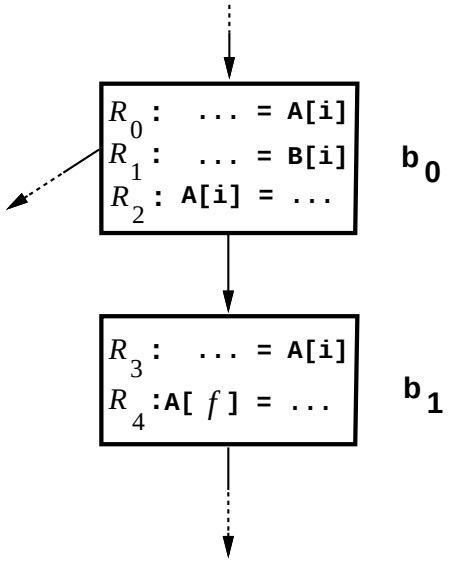
COUT^w(b): Set of data that conditionally downwardly reach the end of block b from write references.

Table I: Sets of downwardly reaching data (and references, where appropriate) that are computed during forward flow analysis.

techniques work well when propagating information at a name-only level, interval analysis is needed to preserve subscript information when propagating reaching information across loops. Therefore, we use a variant of interval analysis that was devised by Gross and Steenkiste [GS90] specifically for combining flow and dependence analyses. Forward flow analysis is used to compute downwardly reaching references; backward flow analysis is used to compute upwardly reaching ones. The reasons for restricting ourselves to unconditional redundancies are twofold: first, the overhead for eliminating conditional redundancies is likely to outweigh the benefits, especially in the parallel case where non-local communication might be needed to detect whether a redundancy actually exists. Second, restricting ourselves to detecting unconditional redundancies substantially simplifies the detection algorithm.

During forward flow analysis, the sets from Table I are computed for each basic block b. In contrast to classical flow analysis, both unconditional sets and their conditional counterparts are computed for each basic block. Although ultimately we are interested in information regarding unconditionally reaching data only, computing the *write subsets* of the conditionally reaching sets provides array kill information that is needed to conservatively compute the unconditionally reaching sets. A write subset, denoted by the superscript w, contains the subset of data from the original set that reach from write references. During forward flow analysis, all the write subsets are used exclusively to generate kill information. Hence, to conserve space, these write subsets store data information only (i.e., $\mathcal{VAR}$ and $\mathcal{SHAPE}$) as discussed in Section 4.1; reference information (i.e., $\mathcal{REF}$) is unnecessary. After flow analysis, UIN(b) includes the set of data that unconditionally downwardly reach the beginning of block b and the set of references from which these data reach. From UIN(b), the set of redundancies in block b can be computed.

As an example of computing these sets for straight-line code, consider Figure 8. UREF(b_0) contains the data that reach the end of block b_0 from references within b_0. Therefore, UREF(b_0) contains two data descriptors: the first is for the datum `B[i]` and the reference list consisting of R_1 and the second is for the datum `A[i]` and the reference list consisting of R_2. Reference R_0 is excluded from the reference list associated with `A[i]`



$$
\begin{aligned}
\mathsf{UIN}(b_0) &= \phi \\
\mathsf{UREF}(b_0) &= \{\ \{\texttt{B[i]}, \{R_1\}\},\ \{\texttt{A[i]}, \{R_2\}\}\ \} \\
\mathsf{UOUT}(b_0) &= \{\ \{\texttt{B[i]}, \{R_1\}\},\ \{\texttt{A[i]}, \{R_2\}\}\ \} \\
\mathsf{CIN}^w(b_0) &= \phi \\
\mathsf{CREF}^w(b_0) &= \{\ \texttt{A[i]}\ \} \\
\mathsf{COUT}^w(b_0) &= \{\ \texttt{A[i]}\ \}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{UIN}(b_1) &= \{\ \{\texttt{B[i]}, \{R_1\}\},\ \{\texttt{A[i]}, \{R_2\}\}\ \} \\
\mathsf{UREF}(b_1) &= \{\ \{\texttt{A[i]}, \bot\}\ \} \\
\mathsf{UOUT}(b_1) &= \{\ \{\texttt{B[i]}, \{R_1\}\},\ \{\texttt{A[i]}, \bot\}\ \} \\
\mathsf{CIN}^w(b_1) &= \{\ \texttt{A[i]}\ \} \\
\mathsf{CREF}^w(b_1) &= \{\ \texttt{A}[*]\ \} \\
\mathsf{COUT}^w(b_1) &= \{\ \texttt{A}[*]\ \}
\end{aligned}
$$

Figure 8: Sets of downwardly reaching data (and references, where appropriate) computed for blocks b_0 and b_1 under the assumption that nothing reaches the start of block b_0.

because the value of `A[i]` read at R_0 differs from that at the end of block b_0. $\mathsf{CREF}^w(b_0)$ includes `A[i]`, the only datum written during block b_0. Assuming that nothing reaches the start of block b_0, $\mathsf{UIN}(b_0)$ is empty, so $\mathsf{UOUT}(b_0) = \mathsf{UREF}(b_0)$. Based on this same assumption, $\mathsf{CIN}^w(b_0)$ is also empty, so $\mathsf{COUT}^w(b_0) = \mathsf{CREF}^w(b_0)$.

$\mathsf{UIN}(b_1)$ contains the set of data and references that reach the start of block b_1. In this case, this set is identical to the one that reaches the end of block b_0, so $\mathsf{UIN}(b_1) = \mathsf{UOUT}(b_0)$. Similarly, $\mathsf{CIN}^w(b_1) = \mathsf{COUT}^w(b_0)$.

If f is unknown at compile time, the compiler cannot determine the precise element of `A` that is written at reference R_4. To be safe, the compiler must assume that any element of `A` *could* be written at this point, while none is *guaranteed* to be. Consequently, $\mathsf{CREF}^w(b_1)$ and, hence, $\mathsf{COUT}^w(b_1)$ include every element of `A`, denoted by the range $*$. Moreover, because the compiler cannot determine the precise reference from which elements of `A` reach the end of block b_2, the $\mathcal{REF}$ fields of $\mathsf{UREF}(b_2)$ and $\mathsf{UOUT}(b_2)$ contain only the dummy reference $\bot$. The issue of estimation is addressed more thoroughly in Section 4.2.3.

From the flow sets, the set of redundancies can be determined. For example, because $\mathsf{UIN}(b_1)$ includes a data descriptor representing the datum `A[i]` and the reference R_2, `A[i]` reaches reference R_3 from reference R_2. Thus, the read of `A[i]` at R_3 is fully redundant.

While the flow sets can be computed in a single step for the simple two-block example above, interval analysis is needed to handle the general case. In a structured program, two types of intervals arise: those that correspond to loops, and those for which there are no backward branches to blocks within the interval [RP86, GS90]. These are termed *loop intervals* and *non-loop intervals*, respectively. By definition [ASU86, RP86], each interval has one entry block. For structured programs, the program flow graph can be constructed such that each loop interval has not only one entry block but one exit block as well. For example, see Figure 9, where the loop interval consists of basic blocks b_1, b_2, and b_3. The significance of this property of loop intervals will become apparent later on.

The first phase of flow analysis begins with the innermost loops and proceeds to the outermost. When a loop

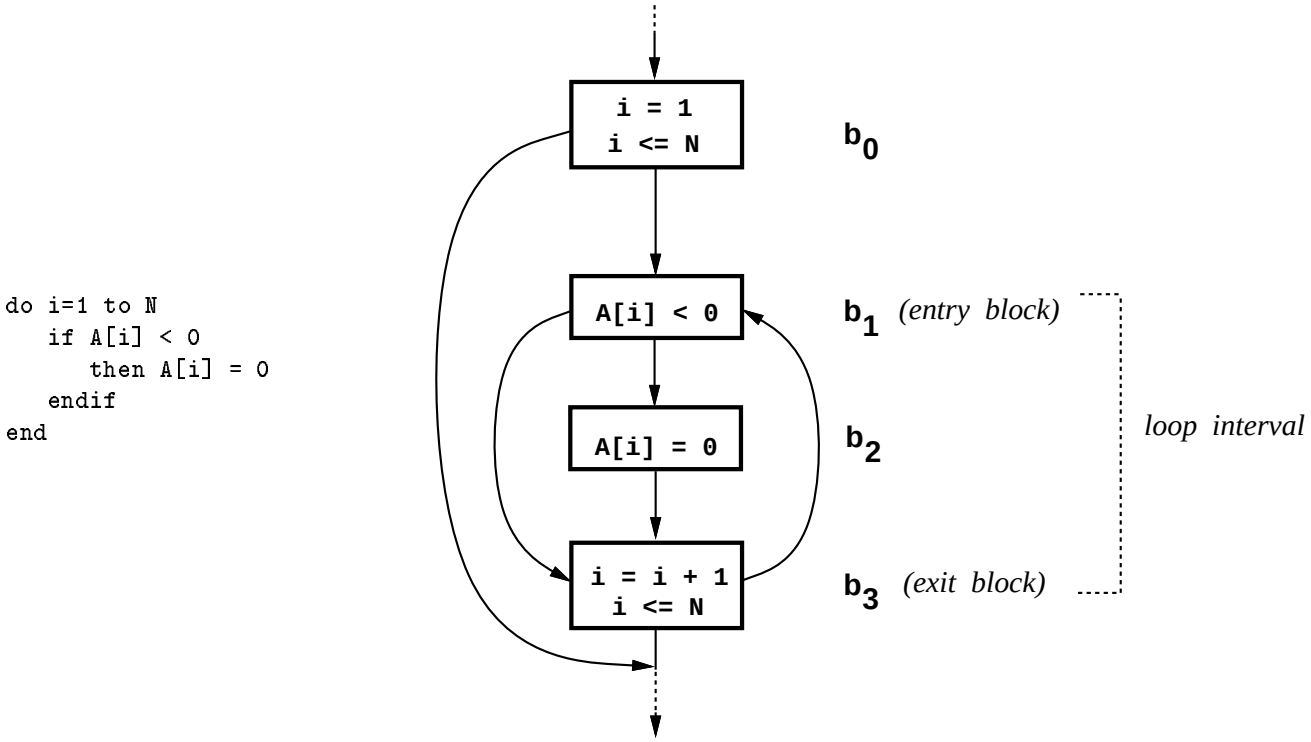


Figure 9: Control flow graph (right) for loop (left). The loop interval, composed of blocks b_2, b_3, and b_4, has a single entry block b_2 and a single exit block b_4.

is processed, interval analysis is performed on the loop. Once the loop is reduced to a single interval, the loop is replaced by a summary block. Once the program is reduced to a single summary block, the second phase of the analysis begins, and summary blocks are replaced by the intervals they represent in the reverse order of that in which the summary blocks were created. As each is replaced, the set information is propagated through the blocks of the interval.

4.2.1 First Phase

During the first phase of analysis, the sets $\mathsf{UREF}(b)$ and $\mathsf{CREF}^w(b)$ are computed directly for each block b. The sets $\mathsf{UIN}(entry\_blk)$ and $\mathsf{CIN}^w(entry\_blk)$ are assumed to be empty. For a non-entry block b, $\mathsf{UIN}(b)$ includes only data that reach the end of *all* predecessor blocks, and the references associated with these data. More formally,

$$\mathsf{UIN}(b) = \bigcap_{b_j \in pred(b)} \mathsf{UOUT}(b_j),$$

where $pred(b)$ is the set of predecessor blocks for b. Meanwhile, $\mathsf{CIN}^w(b)$ includes the data that reach the end of *any* predecessor block:

$$\mathsf{CIN}^w(b) = \bigcup_{b_j \in pred(b)} \mathsf{COUT}^w(b_j).$$

The computation of sets $\mathsf{UOUT}(b)$ and $\mathsf{COUT}^w(b)$ is independent of whether block b is an entry block. When

computing $\mathsf{UOUT}(b)$, block b's conditionally defined write subset $\mathsf{CREF}^w(b)$ is used to filter out read and write references from which data reach the start of block b but not the end, due to writes within b itself. For example, recall Figure 8, where the write at reference R_4 in block b_1 causes reference R_2 to be filtered out of the reference list associated with `A[i]`.

After this filtering step, we add to block b the data referenced within the block itself and the references within this block from which these data reach. More formally,

$$\mathsf{UOUT}(b) = \left(\mathsf{UIN}(b) \mathbin{—}_R \mathsf{CREF}^w(b)\right) \bigcup \mathsf{UREF}(b).$$

In the above equation, note that $\mathsf{CREF}^w(b)$ is used only to filter out references, not data, since data that no longer reach from outside now reach from writes within block b. $\mathsf{COUT}^w(b)$ is computed similarly, with one exception: since no reference information is stored, there are no references to filter out. Therefore,

$$\mathsf{COUT}^w(b) = \mathsf{CIN}^w(b) \bigcup \mathsf{CREF}^w(b).$$

After set information is propagated through all the blocks of an interval I, the entire interval is replaced by a summary block. Assume that I is a non-loop interval and that $b_j \in exit(I)$, where $exit(I)$ is the set of exit blocks for interval I. Because nothing is assumed to reach the entry block of an interval during the first phase, $\mathsf{UOUT}(b_j)$ includes data that reach along all paths from the start of the entry block of interval I to the end of the block b_j, as well as the references from which these data reach. $\mathsf{UREF}(summary\_blk)$ must include the data that unconditionally reach the end of interval I from references within I as well as the references themselves. Observe that this latter set is simply the intersection of those sets that unconditionally reach the end of *each* exit block b_j:

$$\mathsf{UREF}(summary\_blk) = \bigcap_{b_j \in exit(I)} \mathsf{UOUT}(b_j).$$

Meanwhile, $\mathsf{COUT}^w(b_j)$ includes data that reach along some path from the start of the entry block of interval I to end of the block b_j. To ensure that $\mathsf{CREF}^w(summary\_blk)$ includes data that conditionally reach the end of interval I from within I, data that reach the end of *any* exit block b_j are included:

$$\mathsf{CREF}^w(summary\_blk) = \bigcup_{b_j \in exit(I)} \mathsf{COUT}^w(b_j).$$

Loop intervals are summarized using a separate set of flow equations. Recall that a loop interval has one exit block. Assume that the loop iterates from 1 to $lmax$ and that the subscript l is used to denote a set corresponding to iteration l. At this stage, $\mathsf{UOUT}_l(exit\_blk)$ includes data that unconditionally reach the end of the lth loop iteration from reference instances executed during that iteration and the references from which these data reach. $\mathsf{COUT}^w_l(exit\_blk)$ includes data that conditionally reach the end of the lth loop iteration from reference instances executed during that iteration.

A datum reaches the end of a loop from a reference R contained within the loop interval if and only if the following two conditions are met. First, there must exist an iteration l during which the datum is referenced at R. Second, the datum must not be overwritten later during iteration l nor during any successive iteration. $\mathsf{UREF}(summary\_blk)$ contains all data unconditionally referenced during execution of the loop and the references

from which these data reach:

$$\mathsf{UREF}(summary\_blk) = \biguplus_{1 \le l \le lmax} \left[\mathsf{UOUT}_l(exit\_blk) \mathbin{—}_R \biguplus_{l < l' \le lmax} \mathsf{COUT}^w_{l'}(exit\_blk) \right].$$

Meanwhile, $\mathsf{CREF}^w(summary\_blk)$ contains all data conditionally referenced within the loop:

$$\mathsf{CREF}^w(summary\_blk) = \biguplus_{1 \le l \le lmax} \mathsf{COUT}^w_l(exit\_blk).$$

The union operators in the above equations are inscribed with a plus sign (+) to indicate that these unions $\biguplus_{i_{min} \le i \le i_{max}} h(i)$ are implemented by *translation* [Bal90], whereby the range of the union iteration space $i_{min} : i_{max}$ is substituted for the union iteration variable i in function h. Therefore, the time to perform such an operation is independent of the size of the inscribed union's iteration space.

4.2.2 Second Phase

After the first phase of flow analysis is finished, the computation of sets $\mathsf{UREF}(b)$ and $\mathsf{CREF}^w(b)$ is complete. However, the computation of the remaining sets is not complete until the end of the second phase. During this second phase, summary blocks are replaced by the intervals they represent by propagating the sets that reach the start of each summary block through the blocks of the associated interval. For a non-loop interval, the sets that reach its entry block are the same as those that reach the summary block itself:

$$\mathsf{UIN}(entry\_blk) = \mathsf{UIN}(summary\_blk)$$
$$\mathsf{CIN}^w(entry\_blk) = \mathsf{CIN}^w(summary\_blk).$$

For a loop interval, the data that reach the entry block during a given iteration l are the union of those that reach the start of the loop and those that reach the end of any preceding iteration. Since reference information is stored with unconditionally reaching data, the computation of $\mathsf{UIN}_l(entry\_blk)$ further requires the filtering out of every reference R from which data no longer unconditionally reach due to an intervening write of similar data between the earlier instance of R and the start of iteration l.

$$\mathsf{UIN}_l(entry\_blk) = \left[\mathsf{UIN}(summary\_blk) \mathbin{—}_R \biguplus_{1 \le l'' < l} \mathsf{COUT}^w_{l''}(exit\_blk) \right] \bigcup \left[\biguplus_{1 \le l' < l} \left(\mathsf{UOUT}_{l'}(exit\_blk) \mathbin{—}_R \biguplus_{l' < l'' < l} \mathsf{COUT}^w_{l''}(exit\_blk) \right) \right]$$

Since no reference information is stored with $\mathsf{CIN}^w_l(entry\_blk)$, no filtering is necessary:

$$\mathsf{CIN}^w_l(entry\_blk) = \mathsf{CIN}^w(summary\_blk) \bigcup \left[\biguplus_{1 \le l' < l} \mathsf{COUT}^w_{l'}(exit\_blk) \right].$$

Once reaching information is propagated to the entry block of an interval, it can be propagated through the blocks composing the interval using the same flow equations as in the first phase.

Once the second phase is finished, $\mathsf{UIN}(b)$ contains the data that unconditionally downwardly reach the beginning of block b and the references, when known, from which these data reach. $\mathsf{UIN}(b)$ can now be used to compute the two categories of unconditionally redundant reads. Let data descriptor D describe the region of X referenced at R in block b, and let D_u represent the region of X included in $\mathsf{UIN}(b)$. If R is a read reference, then

- R is fully redundant if and only if $\mathcal{SHAPE}(D) \subseteq \mathcal{SHAPE}(D_u)$, whereas
- R is partially redundant if and only if $\mathcal{SHAPE}(D) \bigcap \mathcal{SHAPE}(D_u) \neq \phi$.

4.2.3 Guidelines for Estimation

Lack of information, space limitations, or time limitations may make it impossible to compute the above sets with complete accuracy at compile time. For correctness, we must underestimate reaching information when necessary. Reaching information includes the sets UIN, UOUT, and UREF. Conversely, we must overestimate kill sets, namely CIN^w, CREF^w, and COUT^w, when necessary. Because there are trivial defaults (everything or nothing), clearly we can always do this.

4.3 Algorithm Extensions for Handling Parallel Programs

Recall from Section 3.2 that the definition of reaching for sequential programs must be extended to handle parallel programs, but the definition of redundancy remains unchanged. Therefore, we must make two changes to adapt the algorithm from Section 4.2 to detect read redundancies in parallel programs. First, we must parameterize the computed sets by the processor id p, similar to the parameterization by loop indices discussed earlier. Second, we must prevent data from reaching across an epoch boundary on processor p if the data might have been modified by processor $p' \neq p$ during the epoch terminating at this boundary. After flow analysis, the identification of redundancies proceeds as before.

To demonstrate the need for these two changes, consider the situation for processor $p = p_i$. (Recall the scheduling assumptions that iteration i is executed by processor p_i and that $p_{seq} = p_1$.) In Figures 5 and 6, there are two parallel epochs (corresponding to the first two parallel loops), followed by a sequential epoch, and then another parallel epoch (corresponding the third parallel loop).

In Figure 5, `A[i]` is added to processor p's reaching set during the first epoch. Because `A[i]` is not modified by any other processor $p' \neq p$ during any of the first three epochs, the value referenced in the first epoch is still current in the last epoch. Therefore, `A[i]` reaches R_4 from R_0 across all three epoch boundaries, so the read of `A[i]` at R_4 by processor p is redundant.

In Figure 6, `B[i]` is added to processor p's reaching set during the first epoch. Because `B[i]` is (trivially) not modified by any other processor $p' \neq p$ during the first epoch, `B[i]` reaches across the epoch boundary to the start of the second epoch. However, unlike the situation in Figure 5, `B[i]` is written by a processor $p' \neq p$ during either the second or third epoch. Therefore, processor p's copy is stale at the of the epoch where this write occurs, so `B[i]` must be removed from processor p's reaching set before crossing the boundary into the next epoch. Because `B[i]` does not reach R_4, the read of `B[i]` at R_4 by processor p is not redundant.

4.3.1 Processing Boundary Blocks

The representation of a doall loop in the control flow graph for a program P is similar to the representation for non-doall loops, except that there is no backward edge emanating from the end of the loop back to the beginning. Each doall statement and enddoall statement corresponds to a basic block in P, collectively referred to as *boundary blocks* since each represents a boundary between two epochs. (If two doall loops are always executed in succession, then the boundary blocks for representing the enddoall statment of the first doall loop and the one for representing the doall statement of the next doall loop can be trivially merged into one boundary block for analysis.)

The computation of $\mathsf{UIN}_p(boundary\_blk)$ and $\mathsf{CIN}_p^w(boundary\_blk)$ is the same as for any other block type. $\mathsf{UOUT}_p(boundary\_blk)$ and $\mathsf{COUT}_p^w(boundary\_blk)$ contain data that reach the start of the boundary block on

processor p and are not written by processor p', $p' \neq p$, during the epoch terminating at this boundary block. More formally, let $\mathsf{KILL\_IN}^w_{p'}(boundary\_blk)$ contain the data that are written by processor p' during the epoch terminating at this boundary block. Then $\mathsf{UOUT}_p(boundary\_blk)$ and $\mathsf{COUT}^w_p(boundary\_blk)$ are computed as follows:

$$\mathsf{UOUT}_p(boundary\_blk) = \mathsf{UIN}_p(boundary\_blk) - \biguplus_{p' \neq p} \mathsf{KILL\_IN}^w_{p'}(boundary\_blk)$$
$$\mathsf{COUT}^w_p(boundary\_blk) = \mathsf{CIN}^w_p(boundary\_blk) - \biguplus_{p' \neq p} \mathsf{KILL\_IN}^w_{p'}(boundary\_blk).$$

The above two equations can be simplified further when the boundary block corresponds to a doall statement. Because the epoch that ends at this boundary block must be a sequential epoch, at most one processor (processor p_{seq}) could perform any writes during the terminating epoch. Therefore, whenever the boundary block represents a doall statement,

$$\biguplus_{p' \neq p} \mathsf{KILL\_IN}^w_{p'}(boundary\_blk) = \begin{cases} \mathsf{KILL\_IN}^w_{p_{seq}}(boundary\_blk) & \text{if } p \neq p_{seq} \\ \phi & \text{if } p = p_{seq}. \end{cases}$$

As an example of processing boundary blocks, consider the code in Figure 6 which consists of four epochs. There are three epoch boundaries, after the first, second and third epochs, which are numbered 1, 2 and 3, respectively. Assume that $p = p_i$ and recall that $p_{seq} = p_1$. For each boundary block, the sets UIN, UOUT and $\mathsf{KILL\_IN}^w$ are shown below.

$$\mathsf{UIN}_p(boundary\_blk\ 1) = \{\ \{\mathtt{B[i]}, \{R_0, R_1\}\}\ \}$$
$$\biguplus_{p' \neq p} \mathsf{KILL\_IN}^w_{p'}(boundary\_blk\ 1) = \{\ \mathtt{B[1 : i-1]},\ \mathtt{B[i+1 : N]}\ \}$$
$$\mathsf{UOUT}_p(boundary\_blk\ 1) = \{\ \{\mathtt{B[i]}, \{R_0, R_1\}\}\ \}$$

$$\mathsf{UIN}_p(boundary\_blk\ 2) = \{\ \{\mathtt{B[i]}, \{R_0, R_1\}\},\ \{\mathtt{B[i-1]}, \{R_2\}\}\ \}$$
$$\biguplus_{p' \neq p} \mathsf{KILL\_IN}^w_{p'}(boundary\_blk\ 2) = \{\ \mathtt{B[0 : i-2]},\ \mathtt{B[i : N-1]}\ \}$$
$$\mathsf{UOUT}_p(boundary\_blk\ 2) = \begin{cases} \{\ \{\mathtt{B[i-1]}, \{R_2\}\}\ \} & \text{if } p \neq p_N \\ \{\ \{\mathtt{B[N]}, \{R_0, R_1\}\},\ \{\mathtt{B[N-1]}, \{R_2\}\}\ \} & \text{if } p = p_N \end{cases}$$

$$\mathsf{UIN}_p(boundary\_blk\ 3) = \begin{cases} \{\ \{\mathtt{B[0]}, \{R_2\}\},\ \{\mathtt{B[N]}, \{R_3\}\}\ \} & \text{if } p = p_1 \\ \{\ \{\mathtt{B[i-1]}, \{R_2\}\}\ \} & \text{if } p \neq p_1 \text{ and } p \neq p_N \\ \{\ \{\mathtt{B[N]}, \{R_0, R_1\}\},\ \{\mathtt{B[N-1]}, \{R_2\}\}\ \} & \text{if } p = p_N \end{cases}$$
$$\biguplus_{p' \neq p} \mathsf{KILL\_IN}^w_{p'}(boundary\_blk\ 3) = \begin{cases} \phi & \text{if } p = p_1 \\ \{\ \mathtt{B[N]}\ \} & \text{if } p \neq p_1 \end{cases}$$
$$\mathsf{UOUT}_p(boundary\_blk\ 3) = \begin{cases} \{\ \{\mathtt{B[0]}, \{R_2\}\},\ \{\mathtt{B[N]}, \{R_3\}\}\ \} & \text{if } p = p_1 \\ \{\ \{\mathtt{B[i-1]}, \{R_2\}\}\ \} & \text{if } p \neq p_1 \end{cases}$$

Therefore, $\mathsf{UIN}_p(R_4) = \mathsf{UOUT}_p(boundary\_blk\ 3)$, and $\mathsf{UIN}_p(R_4) \bigcap \mathsf{UREF}_p(R_4) = \phi$, so reference R_4 is not redundant.

$\mathsf{KILL\_REF}_p^w(b)$: Set of data that might be written by processor p within block b during epoch instance e_{exit}.

$\mathsf{KILL\_IN}_p^w(b)$: Set of data that might be written by processor p before the start of block b during epoch instance e_{entry}.

$\mathsf{KILL\_OUT}_p^w(b)$: Set of data that might be written by processor p before the end of block b during epoch instance e_{exit}.

$\mathsf{PREV\_REF}_p^w(b)$: Set of data that might be written by processor p within block b prior to epoch instance e_{exit}.

$\mathsf{PREV\_IN}_p^w(b)$: Set of data that might be written by processor p prior to epoch instance e_{entry}.

$\mathsf{PREV\_OUT}_p^w(b)$: Set of data that might be written by processor p prior to epoch instance e_{exit}.

Table II: For parallel programs, in addition to the reaching sets from Table I, the above reaching sets must be computed for each block b. e_{entry} is the epoch instance that is current on entrance to b, and e_{exit} is the epoch instance that is current on exit to b. (Unless b is a boundary block or a block summarizing an interval containing a boundary block, $e_{entry} = e_{exit}$.)

4.3.2 Computing $\mathsf{KILL\_IN}^w$, $\mathsf{KILL\_OUT}^w$, and $\mathsf{KILL\_REF}^w$

Intuitively, $\mathsf{KILL\_IN}_p^w(b)$ tracks data associated with writes by processor p that might kill reaching data on processor $p' \neq p$. To compute $\mathsf{KILL\_IN}_p^w(b)$ correctly for boundary blocks, it must be computed for every other block as well. Moreover, the corresponding $\mathsf{KILL\_OUT}_p^w(b)$ and $\mathsf{KILL\_REF}_p^w(b)$ from Table II are also needed. Since $\mathsf{KILL\_IN}_p^w(b)$, $\mathsf{KILL\_OUT}_p^w(b)$, and $\mathsf{KILL\_REF}_p^w(b)$ are used exclusively to elicit kill information, it is only necessary to keep track of data, not references.

More formally, let e_{entry} be the epoch instance that is current on entrance to block b. Let e_{exit} be the epoch instance that is current on exit from block b. Note that epoch instance e_{entry} is the same epoch instance that is current on exit to each of block b's predecessors. Moreover, unless b is a boundary block or a block summarizing an interval containing a boundary block, $e_{entry} = e_{exit}$. $\mathsf{KILL\_IN}_p^w(b)$ contains data that might be written by processor p before the start of block b during epoch instance e_{entry}, $\mathsf{KILL\_OUT}_p^w(b)$ contains data that might be written by processor p before the end of block b during epoch instance e_{exit}, and $\mathsf{KILL\_REF}_p^w(b)$ contains data that might be written by processor p within block b during e_{exit}.

For example, consider the computation of $\mathsf{KILL\_REF}_p^w(summary\_blk)$ for the intervals that represent the outer loops in Figures 10 and 11. In Figure 10, every path through the j loop *must* cross a boundary block. Therefore, epoch instance e_{entry} ends at the doall statement executed during iteration $j = 1$. Epoch instance e_{exit} starts after the execution of the enddoall statement during iteration $j = N$. The only write that occurs during block *summary_blk* during epoch instance e_{exit} is the write of `B[j]` at R_1 by processor p_{seq} during iteration $j = N$. Therefore,

$$\mathsf{KILL\_REF}_p^w(summary\_blk) = \begin{cases} \{\ \texttt{B[N]}\ \} & \text{if } p = p_{seq} \\ \phi & \text{otherwise.} \end{cases}$$

```
    shared A,B

    do j=1 to N
       doall i=1 to N
R0 :      A[i] = ···
       enddoall
R1 :   B[j] = ···
    enddo
```

Figure 10: The interval representing the j loop consists of more than one epoch. All paths through the j loop *must* cross boundary blocks.

```
    shared C,D

    do j=1 to N
       if ···
            then
               doall i=1 to N
R0 :              C[i] = ···
               enddoall
       endif
R1 :   D[j] = ···
    enddo
```

Figure 11: Even though the interval representing the j loop consists of more than one epoch, the j loop *might* still be executed in entirety without crossing a single boundary block.

In Figure 11, there is a path through the j loop that does not cross a boundary block. Therefore, it is possible that $e_{entry} = e_{exit}$. Consequently, $\mathsf{KILL\_REF}_p^w(summary\_blk)$ contains all the data that might be written during epoch instance e_{exit}.

$$\mathsf{KILL\_REF}_p^w(summary\_blk) = \begin{cases} \{\ \texttt{D[1 : N]}\ \} & \text{if } p = p_{seq} \\ \phi & \text{otherwise.} \end{cases}$$

The computation of these three KILL sets proceeds as follows. During the first phase, the $\mathsf{KILL\_IN}_p^w(entry\_blk)$ is initialized to the empty set. For the remaining blocks, both basic and summary blocks, the data that might be written by processor p during this epoch prior to the start of block b are the union of those that might be written during this epoch prior to the end of *any* of block b's predecessors:

$$\mathsf{KILL\_IN}_p^w(b) = \bigcup_{b_j \in pred(b)} \mathsf{KILL\_OUT}_p^w(b_j).$$

The above equation is also used during the second phase except when unsummarizing intervals.

When unsummarizing a non-loop interval, the epoch that is current on entrance to the interval's entry block is the same as that that is current on entry to its summary block:

$$\mathsf{KILL\_IN}_p^w(entry\_blk) = \mathsf{KILL\_IN}_p^w(summary\_blk).$$

When unsummarizing loop intervals, there are two cases. First consider the case where *every* path through the loop body crosses an epoch boundary, as in the loop nest in Figure 10. In this case, the epoch that is current on entrance to iteration $l > 1$ cannot be the one that is current on entrance to iteration $l-1$. Therefore, $\mathsf{KILL\_IN}_{p,l}^w(entry\_blk)$ only contains data referenced during iteration $l-1$ that might be written during the epoch instance that is current on exit to iteration $l-1$:

$$\mathsf{KILL\_IN}_{p,l}^w(entry\_blk) = \mathsf{KILL\_OUT}_{p,l-1}^w(exit\_blk).$$

On the first iteration (i.e., $l = 1$), trivially,

$$\mathsf{KILL\_IN}_{p,1}^w(entry\_blk) = \mathsf{KILL\_IN}_p^w(summary\_blk).$$

In the second case, at least one path does *not* cross an epoch boundary, as in the loop nest in Figure 11. Therefore, $\mathsf{KILL\_IN}_{p,l}^w(entry\_blk)$ must also include all data that *might* have been written during the same epoch instance during earlier iterations.

$$\mathsf{KILL\_IN}_{p,l}^w(entry\_blk) = \left[\biguplus_{1 \le l' < l} \mathsf{KILL\_OUT}_{p,l'}^w(exit\_blk)\right] \bigcup \mathsf{KILL\_IN}_p^w(summary\_blk).$$

Consider the computation of $\mathsf{KILL\_REF}_p^w(b)$. If b is a basic block, $\mathsf{KILL\_REF}_p^w(b)$ contains all data conditionally written during the block:

$$\mathsf{KILL\_REF}_p^w(b) = \mathsf{CREF}_p^w(b).$$

Otherwise, b is a summary block for an interval I. If I is a non-loop interval, then $\mathsf{KILL\_REF}_p^w(b)$ contains all data written along any path during the epoch instance that is current on exit to b:

$$\mathsf{KILL\_REF}_p^w(b) = \bigcup_{b_j \in exit(I)} \mathsf{KILL\_OUT}_p^w(b_j).$$

If I is a loop interval, then the computation of $\mathsf{KILL\_REF}_p^w(b)$ depends on whether every path through I crosses an epoch boundary. If every path crosses an epoch boundary, $\mathsf{KILL\_REF}_p^w(b)$ cannot include any data from iterations other than the last iteration, so

$$\mathsf{KILL\_REF}_p^w(b) = \mathsf{KILL\_OUT}_{p,lmax}^w(exit\_blk).$$

If some path does not cross an epoch boundary, the epoch that is current on exit to b might be the same epoch that is current on entrance to b. Therefore, $\mathsf{KILL\_REF}_p^w(b)$ can include data written during any iteration as long

as the epoch during which they are written is current at the end of that iteration. More formally,

$$\mathsf{KILL\_REF}_p^w(b) = \biguplus_{1 \le l \le lmax} \mathsf{KILL\_OUT}_{p,l}^w(exit\_blk).$$

As with other REF sets, the computation of this set is complete after the first phase of flow analysis.

Consider the computation of $\mathsf{KILL\_OUT}_p^w(b)$, which includes all data that might be written during epoch e_{exit}. If b is a basic block other than a boundary block, then

$$\mathsf{KILL\_OUT}_p^w(b) = \mathsf{KILL\_IN}_p^w(b) \bigcup \mathsf{KILL\_REF}_p^w(b).$$

If b is a boundary block, then

$$\mathsf{KILL\_OUT}_p^w(b) = \phi.$$

Otherwise, b is a summary block and the computation of $\mathsf{KILL\_OUT}_p^w(b)$ depends on whether every path through b crosses an epoch boundary. If every path through b crosses an epoch boundary, e_{entry} cannot be the same as e_{exit}. Therefore, $\mathsf{KILL\_OUT}_p^w(b)$ only includes data written during block b during e_{exit}:

$$\mathsf{KILL\_OUT}_p^w(b) = \mathsf{KILL\_REF}_p^w(b).$$

If some path through b does not cross an epoch boundary, then e_{entry} might be the same as e_{exit}. In this case, the computation of $\mathsf{KILL\_OUT}_p^w(b)$ is the same as if b were a (non-boundary) basic block:

$$\mathsf{KILL\_OUT}_p^w(b) = \mathsf{KILL\_IN}_p^w(b) \bigcup \mathsf{KILL\_REF}_p^w(b).$$

4.3.3 Computing PREV_IN^w, PREV_OUT^w, and PREV_REF^w

Section 4.3.4 discusses the extensions for computing UIN, UOUT, UREF, CIN^w, COUT^w, and CREF^w in the presence of doall loops. However, before we can present these extensions, we need the PREV sets: PREV_IN^w, PREV_OUT^w, and PREV_REF^w. Intuitively, these three sets provide additional kill information that is needed to track data associated with potentially intervening write references. $\mathsf{PREV\_REF}_p^w(b)$ contains data that might be written by processor p within block b prior to epoch instance e_{exit}. $\mathsf{PREV\_IN}_p^w(b)$ contains data that might be written by processor p prior to epoch instance e_{entry}. $\mathsf{PREV\_OUT}_p^w(b)$ contains data that might be written by processor p prior to epoch instance e_{exit}.

As an example, consider Figures 10 and 11 again. Assume that $p_{seq} = p_1$. The PREV_REF sets for the intervals represented by the outer loops are presented below. In Figure 10,

$$\mathsf{PREV\_REF}_{p'}^w(summary\_blk) = \begin{cases} \{\ \mathtt{A[1]},\ \mathtt{B[1:N-1]}\ \} & \text{if } p = p_1 \\ \{\ \mathtt{A[i]}\ \} & \text{otherwise.} \end{cases}$$

In Figure 11,

$$\mathsf{PREV\_REF}_{p'}^w(summary\_blk) = \begin{cases} \{\ \mathtt{C[1]},\ \mathtt{D[1:N-1]}\ \} & \text{if } p = p_1 \\ \{\ \mathtt{C[i]}\ \} & \text{otherwise.} \end{cases}$$

How is $\mathsf{PREV\_REF}_p^w(b)$ computed? If block b is not a summary block, then

$$\mathsf{PREV\_REF}_p^w(b) = \phi.$$

The same equation holds for summary blocks representing intervals that do not contain boundary blocks, because the entire summary block is contained within a single epoch (i.e., $e_{entry} = e_{exit}$).

In the remaining case, block b summarizes an interval that contains a boundary block. If I is a non-loop interval, then

$$\mathsf{PREV\_REF}_p^w(b) = \bigcup_{b_j \in exit(I)} \mathsf{PREV\_OUT}_p^w(b_j).$$

If I is a loop interval, then $\mathsf{PREV\_REF}_p^w(b)$ includes all data that might be written prior to iteration $lmax$, as well as all data that might be written during iteration $lmax$ prior to the epoch instance that is current on exit to iteration $lmax$. We exclude data written during the epoch instance that is current on exit to $lmax$, because this is the same epoch instance that is current on exit to the loop (i.e., on exit to summary block b). More formally,

$$\mathsf{PREV\_REF}_p^w(b) = \left(\biguplus_{1 \le l \le lmax} \mathsf{PREV\_OUT}_{p,l}^w(exit\_blk)\right) \bigcup \left(\biguplus_{1 \le l < lmax} \mathsf{KILL\_OUT}_{p,l}^w(exit\_blk)\right).$$

The computation of this set is complete after the first phase of flow analysis.

Consider the computation of $\mathsf{PREV\_OUT}_p^w(b)$. If b is a basic block, then no boundary block is crossed, so

$$\mathsf{PREV\_OUT}_p^w(b) = \mathsf{PREV\_IN}_p^w(b).$$

For a boundary block b, the set of data written during the epoch ending at this boundary block must be added to $\mathsf{PREV\_OUT}_p^w(b)$, so

$$\mathsf{PREV\_OUT}_p^w(b) = \mathsf{PREV\_IN}_p^w(b) \bigcup \mathsf{KILL\_IN}_p^w(b).$$

For a summary block b, the computation of $\mathsf{PREV\_OUT}_p^w(b)$ depends on whether the interval represented by b includes a boundary block. If it does not, then the computation is the same as that for a (non-boundary) basic block:

$$\mathsf{PREV\_OUT}_p^w(b) = \mathsf{PREV\_IN}_p^w(b).$$

Otherwise, b is a summary block that includes a boundary block, so $\mathsf{PREV\_OUT}_p^w(b)$ includes the data written prior to b or during b, except for those written during epoch instance e_{exit}:

$$\begin{aligned} \mathsf{PREV\_OUT}_p^w(b) \quad = \quad & \mathsf{PREV\_IN}_p^w(b) \bigcup \mathsf{KILL\_IN}_p^w(b) \\ & \bigcup \mathsf{PREV\_REF}_p^w(b). \end{aligned}$$

During the first phase, $\mathsf{PREV\_IN}_p^w(entry\_blk)$ is initialized to the empty set. For a block b that is not an entry block, $\mathsf{PREV\_IN}_p^w(b)$ is trivially the union of the corresponding OUT sets of block b's predecessors:

$$\mathsf{PREV\_IN}_p^w(b) = \bigcup_{b_j \in pred(b)} \mathsf{PREV\_OUT}_p^w(b_j).$$

Unlike the other OUT sets, the computation of $\mathsf{PREV\_OUT}^w$ is complete after the first phase of flow analysis. Since $\mathsf{PREV\_IN}^w$ is used solely to compute $\mathsf{PREV\_OUT}^w$, the computation of $\mathsf{PREV\_IN}^w$ is also complete after the

first phase. The final set of equations for performing forward flow analysis is presented in the Appendix.

4.3.4 Processing Summary Blocks

How is the computation of UIN, UOUT, UREF, CIN^w, COUT^w, and CREF^w affected by the presence of doall loops? If a block is neither a boundary block nor a summary block representing an interval that contains a boundary block, then the flow equations derived earlier in Section 4.2 can be applied directly to propagate information across this block. These same flow equations can also be applied directly to summarize and unsummarize an interval, as long as that interval does not include a boundary block. Since the processing of boundary blocks has been discussed earlier in Section 4.3.1, all that remains when redundancy-preventing write references are present is to extend the flow equations to prevent data from reaching across summary blocks representing intervals that include boundary blocks.

More precisely, let x be a datum that reaches the start of a summary block from a reference by processor p. Assume that the interval represented by a summary block contains at least one boundary block. Let e_{entry} and e_{exit} be defined as before. Clearly, there must be some path from e_{entry} to e_{exit} that crosses an epoch boundary. Let e be an epoch instance, if any, that is completely contained within that summary block and executed between e_{entry} and e_{exit}. Then x reaches across the summary block only if it can be guaranteed that there is no write of x by a processor $p' \neq p$ during either epoch instance e_{entry} or e. Recall that $\mathsf{KILL\_IN}^w_{p'}(summary\_blk)$ contains all the data that might be written by processor p' prior to the start of the summary block, and that $\mathsf{PREV\_REF}^w_{p'}(summary\_blk)$ contains all the data that might be written by processor p' during $summary\_blk$ prior to e_{exit}. Thus the set of data written by a processor other than p prior to e_{exit} can then be computed as

$$\mathsf{REF}^w_{p'} = \left(\mathsf{KILL\_IN}^w_{p'}(summary\_blk) \bigcup \mathsf{PREV\_REF}^w_{p'}(summary\_blk)\right).$$

The computation of $\mathsf{UOUT}_p(summary\_blk)$ and $\mathsf{COUT}^w_p(summary\_blk)$ must be modified to prevent any data contained in the set

$$\biguplus_{p' \neq p} \mathsf{REF}^w_{p'}$$

from reaching across the summary block on processor p. Therefore,

$$\begin{aligned}
\mathsf{UOUT}_p(summary_blk) &= \Big(\mathsf{UIN}_p(summary_blk) - \biguplus_{p' \neq p} \mathsf{REF}^w_{p'}(summary_blk) \\
&\quad - _R \mathsf{CREF}^w_p(summary_blk)\Big) \bigcup \mathsf{UREF}_p(summary_blk) \\
\mathsf{COUT}^w_p(summary_blk) &= \Big(\mathsf{CIN}^w_p(summary_blk) - \biguplus_{p' \neq p} \mathsf{REF}^w_{p'}(summary_blk)\Big) \\
&\quad \bigcup \mathsf{CREF}^w_p(summary_blk).
\end{aligned}$$

Consider the computation of $\mathsf{UREF}_p(summary\_blk)$ and $\mathsf{CREF}^w_p(summary\_blk)$, the sets of data that reach the end of $summary\_blk$. If $summary\_blk$ represents a non-loop interval, the equations from Section 4.2 can be used directly regardless of whether the interval includes a boundary block, because the OUT sets from which these

REF sets are computed already account for any boundary blocks:

$$\begin{aligned}\mathsf{UREF}_p(summary\_blk) &= \bigcap_{b_j \in exit(I)} \mathsf{UOUT}_p(b_j) \\ \mathsf{CREF}^w_p(summary\_blk) &= \bigcup_{b_j \in exit(I)} \mathsf{COUT}^w_p(b_j).\end{aligned}$$

If *summary_blk* represents a loop interval, the flow equations from Section 4.2 must be extended to filter out data referenced during iteration l by processor p that might be written by processor $p' \neq p$ either during loop iteration $l' > l$, or within a later epoch instance during iteration l. Therefore,

$$\begin{aligned}\mathsf{UREF}_p(summary\_blk) &= \biguplus_{1 \le l \le lmax} \Big[\mathsf{UOUT}_{p,l}(exit\_blk) - \biguplus_{p' \neq p, l \le l' < lmax} \mathsf{KILL\_OUT}^w_{p',l'}(exit\_blk) \\ &\quad - \biguplus_{p' \neq p, l < l' \le lmax} \mathsf{PREV\_OUT}^w_{p',l'}(exit\_blk) -_R \biguplus_{l < l' \le lmax} \mathsf{COUT}^w_{p,l'}(exit\_blk)\Big]\end{aligned}$$

$$\begin{aligned}\mathsf{CREF}^w_p(summary\_blk) &= \biguplus_{1 \le l \le lmax} \Big[\mathsf{COUT}^w_{p,l}(exit\_blk) - \biguplus_{p' \neq p, l \le l' < lmax} \mathsf{KILL\_OUT}^w_{p',l'}(exit\_blk) \\ &\quad - \biguplus_{p' \neq p, l < l' \le lmax} \mathsf{PREV\_OUT}^w_{p',l'}(exit\_blk)\Big].\end{aligned}$$

With the exception of the handling of entry blocks associated with loop intervals during the second phase of flow analysis, $\mathsf{UIN}_p(b)$ and $\mathsf{CIN}^w_p(b)$ are computed as before. During the second phase, the flow equations determining $\mathsf{UIN}_{p,l}(entry\_blk)$ and $\mathsf{CIN}^w_{p,l}(entry\_blk)$ from Section 4.2 must be extended to include data referenced at R by processor p before iteration l, only if there is no possibility of an intervening write by a processor $p' \neq p$ between R and the start of iteration l. Therefore, the sets that reach the start of the first loop iteration (i.e., $l = 1$) are trivially the same as those that reach the start of the loop:

$$\begin{aligned}\mathsf{UIN}_{p,l}(entry\_blk) &= \mathsf{UIN}_p(summary\_blk) \\ \mathsf{CIN}^w_{p,l}(entry\_blk) &= \mathsf{CIN}^w_p(summary\_blk).\end{aligned}$$

For $l > 1$, we compute the sets as follows:

$$\begin{aligned}\mathsf{UIN}_{p,l}(entry\_blk) &= \Big(\mathsf{UIN}_p(summary\_blk) - \biguplus_{p' \neq p, 1 \le l'' < l} \mathsf{PREV\_OUT}^w_{p',l''}(exit\_blk) \\ &\quad - \biguplus_{p' \neq p} \mathsf{KILL\_IN}^w_{p'}(summary\_blk) - \biguplus_{p' \neq p, 1 \le l'' < l-1} \mathsf{KILL\_OUT}^w_{p',l''}(exit\_blk) \\ &\quad -_R \biguplus_{1 \le l'' < l} \mathsf{COUT}^w_{p,l''}(exit\_blk)\Big) \\ &\quad \bigcup \Big[\biguplus_{1 \le l' < l} \Big(\mathsf{UOUT}_{p,l'}(exit\_blk) - \biguplus_{p' \neq p, l' < l'' < l} \mathsf{PREV\_OUT}^w_{p',l''}(exit\_blk) \\ &\quad - \biguplus_{p' \neq p, l' \le l'' < l-1} \mathsf{KILL\_OUT}^w_{p',l''}(exit\_blk) -_R \biguplus_{l' < l'' < l} \mathsf{COUT}^w_{p,l''}(exit\_blk)\Big)\Big]\end{aligned}$$

$$\begin{aligned}\mathsf{CIN}^w_{p,l}(entry\_blk) &= \Big(\mathsf{CIN}^w_p(summary\_blk) - \biguplus_{p' \neq p, 1 \le l'' < l} \mathsf{PREV\_OUT}^w_{p',l''}(exit\_blk) \\ &\quad - \biguplus_{p' \neq p} \mathsf{KILL\_IN}^w_{p'}(summary\_blk) - \biguplus_{p' \neq p, 1 \le l'' < l-1} \mathsf{KILL\_OUT}^w_{p',l''}(exit\_blk)\Big) \\ &\quad \bigcup \Big[\biguplus_{1 \le l' < l} \Big(\mathsf{COUT}^w_{p,l'}(exit\_blk) - \biguplus_{p' \neq p, l' < l'' < l} \mathsf{PREV\_OUT}^w_{p',l''}(exit\_blk) \\ &\quad - \biguplus_{p' \neq p, l' \le l'' < l-1} \mathsf{KILL\_OUT}^w_{p',l''}(exit\_blk)\Big)\Big].\end{aligned}$$

4.3.5 Guidelines for Estimation

As in the case of handling sequential programs, we can safely estimate all sets when exact representation is not possible. In general, for correctness, we should not overestimate reaching sets and we should not underes-

timate kill sets. Clearly, the important reaching set is UOUT. To prevent UOUT from being overestimated, we must underestimate the kill sets $\mathsf{KILL\_IN}^w$, $\mathsf{KILL\_OUT}^w$, and $\mathsf{KILL\_REF}^w$, when necessary. If we overestimate $\mathsf{KILL\_IN}^w$, $\mathsf{KILL\_OUT}^w$, and $\mathsf{KILL\_REF}^w$, then the kill sets $\mathsf{PREV\_IN}^w$, $\mathsf{PREV\_OUT}^w$, and $\mathsf{PREV\_REF}^w$ will also be overestimated, which may yield conservative but nonetheless correct results.

Note that, if we overestimate $\mathsf{KILL\_IN}^w$, then we might end up underestimating COUT^w. Recall from Section 4.2.3, that we previously assumed that COUT^w would be overestimated. In actuality, overestimating COUT^w is merely an easy method for ensuring that every datum x that is contained in both the actual set COUT^w and in the estimated set $\mathsf{UOUT}_p(b)$ is also included in the estimated set COUT^w, which we can accomplish even when $\mathsf{KILL\_IN}_p^w(b)$ is overestimated.

4.4 Cost Analysis (Worst Case)

Let $\mathcal{S}$ and $\mathcal{V}$ be the number of statements and variables in a given program. Assume that the upper bounds on the nesting depth, the number of array dimensions, and the number of summary shapes permitted per data descriptor are small constants. Then, the total number of basic and summary blocks is $O(\mathcal{S})$ [RP86]. Consequently, the number of steps is also bounded by $O(\mathcal{S})$. At worst, the time to perform a single set operation is $O(\mathcal{S}\,\mathcal{V})$. Because there are only a finite number of set operations per step, the time cost per step is also $O(\mathcal{S}\,\mathcal{V})$. Therefore, in the worst case, the time cost for the redundancy detection algorithm is

$$O(\#steps \cdot time\_per\_step) = O(\mathcal{S}^2\mathcal{V}).$$

The space required to store the set information associated with a single block is $O(\mathcal{S}\,\mathcal{V})$. Consequently, in the worst case, the space cost for the redundancy detection algorithm is

$$O(\#blocks \cdot space\_per\_block) = O(\mathcal{S}^2\mathcal{V}).$$

5 Other Variations of Redundancy Analysis

In this section, we briefly explain how the algorithm presented in this paper can be adapted to identify write redundancies and redundancy inducers, catch promising classes of conditional redundancies, and aggressively compute ranges in which data can be safely prefetched. We also discuss extensions for handling other parallel constructs and scheduling strategies.

5.1 Detecting Write Redundancies

Identifying redundant write references is roughly analogous to identifying redundant read references. The key distinctions are enumerated below:

- Identification of redundant read references requires information on *downwardly* reaching data, while identification of redundant write references requires information on *upwardly* reaching data. Consequently, backward flow analysis is used instead of forward flow analysis, and the roles of the IN and OUT sets are reversed.

For example, see Figures 5 and 6. Note that what happens before a read determines whether that read is redundant. In contrast, what happens after a write determines whether that write is redundant.

- Although *both* read and write references can induce read redundancies, *only* write references can induce write redundancies. Therefore, only data reaching from write references are of interest.

  As can be seen in Figures 5 and 6, only writes can induce write redundancies. Therefore reads are only of interest when computing kill information.

- When we compute reaching data for the identification of read redundancies, a datum can only reach a reference instance executed on the *same* processor as the reference instance from which it reaches. In contrast, when we compute reaching data for identifying write redundancies, a datum can reach a reference instance executed on a *different* processor.

  For example, in Figure 6, consider the write of `B[i]` at R_0 by processor $p = p_i$. This write reference is redundant because `B[i]` upwardly reaches R_0 on processor p from a write at either R_2 or R_3 by some processor $p' \neq p$.

  Therefore, the equations for computing the sets of data that unconditionally reach across boundary blocks must reflect the fact that data can now reach across boundary blocks from other processors as well.

For specific details regarding algorithm extensions, see Granston [Gra92].

5.2 Identifying Redundancy Inducers

Earlier in this paper, an algorithm was presented for identifying the redundant references themselves, as well as the references that induce each redundancy. The analog in classical flow analysis is use-definition chains [ASU86]. Alternatively, we can compute the analog of definition-use chains, namely the set of redundancy inducers and, for each redundancy inducer, the set of redundancies that it induces.

Consider the problem of identifying read redundancy inducers. First, backward flow analysis is needed to identify the set of data that upwardly reach each read and write reference from later read references. Then we can determine if a read or write reference induces any read redundancies, and if so, the set of redundant reads that it induces.

Meanwhile, forward flow analysis is needed to compute the set of data that downwardly reach each write reference from earlier write references. Then we can determine whether a write reference induces any writes to be redundant, and if so, which ones it induces.

The manner in which the resulting information will be used dictates whether the computation of redundancy-inducer chains or inducer-redundancy chains are more appropriate.

5.3 Catching Promising "Conditional" Redundancies

In general, attempting to detect and eliminate conditional redundancies is unlikely to prove worthwhile. Unless the conditional redundancy occurs frequently in practice, the cost of eliminating the redundancy might outweigh the benefit. Moreover, simply determining when conditional redundancies can be profitably eliminated can be expensive.

```
        while (error > eps)
              do i=1 to N
R0:              ··· = A[i]
              enddo
              ···
              do i=1 to N
R1:              A[i] = ···
              enddo
        endwhile
```

Figure 12: Example of a promising conditional redundancy: R_0 is redundant on all but the first iteration. R_1 is redundant on all but the last iteration. (This example is extracted from a conjugate gradient routine.)

However, there are several commonly occurring cases of conditional redundancies which are indeed worth detecting. In these cases, the conditional redundancy is expected to occur frequently in practice. We present two examples of such cases. The first, depicted in Figure 12, is extracted from a conjugate gradient routine. In this example, the read at R_0 is redundant on all but the first iteration of the while loop. Meanwhile, the write at R_1 is redundant on all but the last iteration of the while loop. Both these conditional redundancies lead to actual redundancies *most* of the time.

The second example, depicted in Figure 13 has been extracted from an independently manually-optimized key routine from a two dimensional cfd code. In this case, the redundancy at R_1 *appears* conditional, since there is a path along which the data does not reach, namely the one from the entrance of the loop to R_1 through the false branch of the if statement. However, this path is never taken. During the first iteration, the true-branch of the if statement is always taken, so the read at R_1 is preceded by a write of the same data at R_0. On all other iterations, the read at R_1 is redundant due the references in the prior iterations. Therefore, the redundancy is really unconditional, but symbolic analysis is needed to detect this case.

In both examples, if we could detect and unroll the preamble or postamble of these loops or both, at least conceptually for analysis purposes, we could detect these redundancies using the algorithm presented earlier in this paper. In fact, actually performing this unrolling would also allow us to eliminate these redundancies without incurring the overhead that would otherwise result from needing to conduct a test on every loop iteration to determine whether a redundancy really exists during that particular iteration.

5.4 Exposing Prefetching Opportunities

Redundancy analysis can also be used to expose prefetching opportunities. In particular, the read references that are detected as non-redundant are the prefetching candidates. By extending the flow analysis to keep track of the references that cause these read references to be stale, we can aggressively compute the ranges in which we can safely insert prefetches for these non-redundant reads.

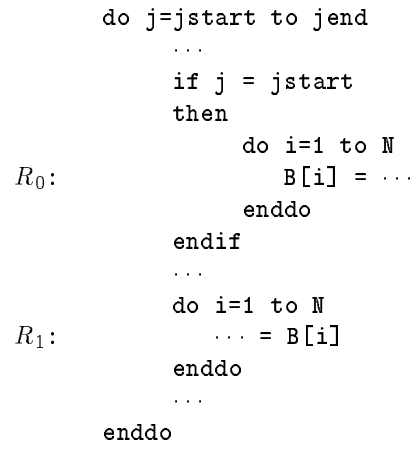


Figure 13: Example of a promising "conditional" redundancy: R_1 is actually an unconditional redundancy that would be detected as conditional. (This example is extracted from an independently manually-optimized key routine of a two dimensional cfd program.)

5.5 Handling Other Parallel Program Models

Suppose that dynamic inter-epoch scheduling is used so that processor reassignments occur at epoch boundaries. In this case, it cannot be established with certainty at compile time whether any potential redundancies that arise from dependences that cross epoch boundaries will exist at run time. Note that the benefits of trying to exploit such opportunities are likely to be slim, whereas the run time cost is likely to be high. Hence, a conservative approach can be used and these details ignored by assuming that references cannot reach across boundaries where processor reassignments occur.

Clearly, the choice between implementing an epoch boundary (a doall or an enddoall statement) with a barrier (static inter-epoch scheduling) or by performing an actual processor reassignment (dynamic inter-epoch scheduling) can be made on a boundary-by-boundary basis. Since the overhead for reassigning processors is generally high, task granularity (where a *task* is a maximal region between processor reassignment boundaries) should be large enough to offset this overhead. Note that static inter-epoch scheduling increases task granularity, thereby reducing overhead and increasing opportunities for redundancy elimination, whereas dynamic inter-epoch scheduling facilitates the balancing of loads between processors. Hence, in general, static inter-epoch scheduling should be used to merge small epochs into reasonably sized tasks, which could then be dynamically scheduled. Redundancy analysis can be performed independently for each program task.

The discussion thus far has focussed on doall-style parallelism. More general classes of parallel constructs allow data to be written on one processor and accessed by another during the same epoch. A simple, conservative technique for handling such programs is to consider all references to data accessed in such a manner as non-redundant. More sophisticated approaches are the subject of future research.

6 Related Work

Traditionally, analysis techniques have either focused on one of two approaches: computing dependences between two references enclosed within a single loop nest or performing precise control flow analysis of scalars in sequential programs while treating arrays at the name-only level. Anderson and Lam [AL93] have extended the first approach so that they could analyze multiple loop nests and, based on this analysis, apply transformations to increase the number of redundancies (equivalently, to increase locality). However, their framework cannot handle conditionally executed code.

Others have concentrated on extending the second approach. Gross and Steenkiste [GS90] and Rosene [Ros90] were the first to combine flow and dependence analyses to analyze array dependences in sequential programs at a more precise level. Cheong and Veidenbaum [CV88], while still treating arrays at a name-only level, were the first to extend flow analysis to handle programs with doall loops. Their technique handles sequential loops and control flow constructs as well. Ferrante et al. [FGS94] presented a combined flow and dependence analysis technique that could be adapted to detect redundancies. While their framework can handle explicitly parallel programs with cobegin/coend and post/wait constructs as well as conditional code and array subscript expressions, it cannot handle loops. In contrast, the combined flow and dependence analysis technique presented in this paper can handle doall loops, sequential loops (including multiple loop nests and non-perfectly nested loops), and conditional code, while considering array subscript expressions.

While we targeted loop-level parallelism, Gupta et al. [GSS94] and Hanxleden and Kennedy [HK93] concentrated on combining flow and dependence analyses with the goal of optimizing the placement of sends and receives when compiling data-parallel programs. They perform their analysis on the sequential version of the program before parallelization and then combine the analysis results with distribution information to determine where communication should be placed. By analyzing the sequential version, they avoid the additional complexities that arise from doall loops. Because our goal is to handle already parallelized programs, we could not apply the same approach.

7 Summary

In general, the problem of identifying potentially redundant references and their inducers is non-trivial. However, when such information is available, it can be used to eliminate redundant memory accesses both in systems that support software-controlled local storage and in cache-based systems. In this paper, we have developed a formal terminology for describing redundant memory accesses and their inducers. We have identified three categories of redundancies and redundancy inducers. We have discussed the limitations of current compiler technology for exposing array redundancies.

We have derived a set of flow equations to be used within an interval-analysis framework for exposing redundant references and those references that induce them. Since performing name-only array analysis is too coarse, and treating arrays at the granularity of individual elements is cost-prohibitive, our algorithm analyzes arrays at the level of regions. The algorithm itself is independent of the method used to summarize the array regions. It is straightforward to adapt existing summary methods for use within our framework. Since it is rare that array regions can be summarized precisely, we have shown how our analysis method can be used even when exact representation of flow sets is not possible. Ours is the first approach to apply combined flow and dependence

analysis to programs with doall style parallelism. Moreover, our technique has the advantage that it can be run after either manual or automatic parallelization. Our approach can also be used to expose prefetching opportunities and identify potentially stale data accesses.

It is important to note that our goal has been to develop an analytical tool for exposing *existing* opportunities to eliminate redundant accesses. The number of opportunities exposed in practice depends on the number that is inherent in the analyzed codes. Consequently, this tool can be used as a yardstick to "measure" reuse opportunities and gauge the effectiveness of specific optimizations.

Clearly, the number of redundancies that exists also depends on the scheduling strategy employed and the information regarding this scheduling strategy that is available to the compiler. When static inter-epoch scheduling is employed, optimizations such as aligning accesses across several doall loops can increase the number of redundancies that exists [GMB]. Note that, even when reuse is detected, insufficient temporal locality or local storage space may prevent us from capitalizing on them. Hence, redundancy analysis is most beneficial when combined with locality-enhancing program transformations.

Acknowledgements

We would like to thank Edward Gornish, Reinhard von Hanxleden, William Jalby, Ken Kennedy, Chuck Koelbel and David Sehr for their feedback on earlier versions, and Debbie Campbell for proofreading this paper. Dr. Granston was supported by a Postdoctoral Research Associateship in Computational Science and Engineering under National Science Foundation Grant No. CDA-9310307, a grant from the International Business Machines Corporation, and the Center for Research on Parallel Computation under Grant No. CCR-9120008. Much of this work was done while Elana Granston was a graduate student at the Center for Supercomputing Research and Development, where support was provided by the Department of Energy under Grant No. DE-FG02-85ER25001 and Cray Research Incorporated. Dr. Veidenbaum was supported by the NASA Ames Research Center under Grant No. NASA NCC 2-559 and the National Science Foundation under Grant No. NSF 89-20891.

References

[AHD93] Bill Applebe, Charles Hardnett, and Sri Doddapaneni. Program Transformation for Locality Using Affinity Regions. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 1993.

[AL93] Jennifer Anderson and Monica Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Programming Languages Design and Implementation*, June 1993.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[Bal90] Vasanth Balasundaram. A Mechanism for Keeping Useful Internal Information in Parallel Programming Tools: The Data Access Descriptor. *Journal of Parallel and Distributed Computing*, 9(2):154–170, June 1990.

[BGM95] François Bodin, Elana D. Granston, and Thierry Montaut. Page-level Affinity Scheduling for Eliminating False Sharing. In *Fifth Workshop on Compilers for Parallel Computers*, Malaga, Spain, June 1995.

[BJEW91] Francois Bodin, William Jalby, Christine Eisenbeis, and Daniel Windheiser. Window-Based Register Allocation. Technical report, INRIA, 1991.

[CKM88] Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe. Automatic Management of Programmable Caches Using Flow Analysis. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 229–238, August 1988.

[CON93] CONVEX Computer Corporation, 3000 Waterview Parkway, Richardson, TX 75083-3851. *Exemplar Architecture*, November 1993. Order No. DHW-014.

[Cra93] Cray Research, Inc. *CRAY T3D System Architecture Overview*, 1993.

[CV88] Hoichi Cheong and Alexander V. Veidenbaum. Stale Data Detection and Coherence Enforcement Using Flow Analysis. In *Proceedings of the International Conference on Parallel Processing*, volume I, pages 138–145, August 1988.

[DMCK92] Ervan Darnell, John M. Mellor-Crummey, and Ken Kennedy. Automatic Software Cache Coherence Through Vectorization. In *Proceedings of the International Conference on Supercomputing*, pages 129–138, July 1992.

[FGS94] Jeanne Ferrante, Dirk Grunwald, and Harini Srinivasan. Computing Communication Sets for Control Parallel Programs. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994.

[GKLS83] Daniel Gajski, David Kuck, Duncan Lawrie, and Ahmed Sameh. Cedar – a Large Scale Multiprocessor. In *Proceedings of the International Conference on Parallel Processing*, pages 524–529, August 1983.

[GMB] Elana D. Granston, Thierry Montaut, and François Bodin. Loop Transformations to Prevent False Sharing. To appear in the *International Journal of Parallel Programming.*

[Gra92] Elana D. Granston. *Reducing Memory Access Delays in Large-Scale, Shared-Memory Multiprocessors.* PhD thesis, Center for Supercomputing Research and Development, Technical Report 1257, University of Illinois at Urbana-Champaign, October 1992.

[GS90] Thomas Gross and Peter Steenkiste. Structured Dataflow Analysis for Arrays and Its Use in an Optimizing Compiler. *Software – Practice & Experience*, 20(2):133–155, February 1990.

[GSS94] Manish Gupta, Edith Schonberg, and Harini Srinivasan. A Unified Framework for Optimizing Communication. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994.

[HHL90] Lorenz Huelsbergen, Douglas Hahn, and James Larus. Exact Data Dependence Analysis Using Data Access Descriptors. Technical Report 945, Computer Science Department, University of Wisconsin-Madison, July 1990.

[HK91] Paul Havlak and Ken Kennedy. An Implementation of Interprocedural Bounded Regular Section Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[HK93] Reinhard von Hanxleden and Ken Kennedy. A Code Placement Framework and Its Application to Communication Generation. Technical Report CRPC-TR93337-S, Center for Research on Parallel Computation, Rice University, October 1993.

[KDCZ94] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed Shared Memory On Standard Workstations and Operating Systems. In *Winter Usenix Conference*, 1994.

[Ken92] Kendall Square Research Corporation. *Kendall Square Research Technical Summary*, 1992.

[LLJ+92] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. In *International Symposium on Computer Architecture*, pages 92–103, May 1992.

[LP92] Z. Lajormi and T. Priol. KOAN: A Shared-Memory for the iPSC/2 Hypercube. In *CONPAR/VAPP92*, LNCS 634. Springer-Verlag, September 1992.

[MR79] E. Morel and C. Renvoise. Global Optimization by Suppression of Partial Redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[Pug92] William Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Communications of the ACM*, pages 102–114, August 1992.

[Ros90] Carl M. Rosene. *Incremental Dependence Analysis*. PhD thesis, Rice University, Technical Report COMP TR90-112, March 1990.

[RP86] Barbara G. Ryder and Marvin C. Paull. Elimination Algorithms for Data Flow Analysis. *Computing Surveys*, 18(3):277–316, September 1986.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global Value Numbers and Redundant Computations. In *ACM Symposium on Principles of Programming Languages*, pages 12–27. ACM, January 1988.

[Sch89] Dale Schouten. An Overview of Interprocedural Analysis Techniques for High Performance Parallelizing Compilers. Master's thesis, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, December 1989.

[TGJ93] Olivier Temam, Elana D. Granston, and William Jalby. To Copy or Not to Copy: A Compile-Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts. In *Supercomputing '93*, pages 410–419, November 1993.

[Vei86] Alexander V. Veidenbaum. A Compiler-assisted Cache Coherence Solution for Multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 1029–1036, August 1986.

Appendix: Equations for Forward Flow Analysis

Basic Equations

Flow equations for propagating reaching information through blocks *other than* entry blocks and boundary blocks during the first and second phases of forward flow analysis. Assume $pred(b)$ is the set of predecessors of block b. (During the first phase of flow analysis, $\mathsf{UIN}_p(entry\_blk)$, $\mathsf{CIN}_p^w(entry\_blk)$, $\mathsf{KILL\_IN}_p^w(entry\_blk)$ and $\mathsf{PREV\_IN}_p^w(entry\_blk)$ are initialized to ϕ.)

$$\mathsf{UIN}_p(b) = \bigcap_{b_j \in pred(b)} \mathsf{UOUT}_p(b_j)$$

$$\mathsf{UOUT}_p(b) = \begin{cases} \Big(\mathsf{UIN}_p(b) \mathbin{—} \biguplus_{p' \neq p} \mathsf{KILL\_IN}_{p'}^w(b) \\ \quad \mathbin{—} \biguplus_{p' \neq p} \mathsf{PREV\_REF}_{p'}^w(b) & \textbf{if} \text{ some execution path through } b \\ \quad \mathbin{—}_R \mathsf{CREF}_p^w(b)\Big) \bigcup \mathsf{UREF}_p(b) & \text{crosses an epoch boundary} \\ \\ \Big(\mathsf{UIN}_p(b) \mathbin{—}_R \mathsf{CREF}_p^w(b)\Big) \bigcup \mathsf{UREF}_p(b) & \textbf{otherwise} \end{cases}$$

$$\mathsf{CIN}_p^w(b) = \bigcup_{b_j \in pred(b)} \mathsf{COUT}_p^w(b_j)$$

$$\mathsf{COUT}_p^w(b) = \begin{cases} \Big[\mathsf{CIN}_p^w(b) \mathbin{—} \biguplus_{p' \neq p} \mathsf{KILL\_IN}_{p'}^w(b) & \textbf{if} \text{ some execution path through } b \\ \quad \mathbin{—} \biguplus_{p' \neq p} \mathsf{PREV\_REF}_{p'}^w(b)\Big] \bigcup \mathsf{CREF}_p^w(b) & \text{crosses an epoch boundary} \\ \\ \mathsf{CIN}_p^w(b) \bigcup \mathsf{CREF}_p^w(b) & \textbf{otherwise} \end{cases}$$

$$\mathsf{KILL\_IN}_p^w(b) = \bigcup_{b_j \in pred(b)} \mathsf{KILL\_OUT}_p^w(b_j)$$

$$\mathsf{KILL\_OUT}_p^w(b) = \begin{cases} \mathsf{KILL\_REF}_p^w(b) & \textbf{if} \text{ every execution path through } b \\ & \text{crosses an epoch boundary} \\ \\ \mathsf{KILL\_IN}_p^w(b) \bigcup \mathsf{KILL\_REF}_p^w(b) & \textbf{otherwise} \end{cases}$$

$$\mathsf{PREV\_IN}_p^w(b) = \bigcup_{b_j \in pred(b)} \mathsf{PREV\_OUT}_p^w(b_j)$$

$$\mathsf{PREV\_OUT}_p^w(b) = \begin{cases} \mathsf{PREV\_IN}_p^w(b) \bigcup \mathsf{KILL\_IN}_p^w(b) & \textbf{if} \text{ some execution path through } b \\ \quad \bigcup \mathsf{PREV\_REF}_p^w(b) & \text{crosses an epoch boundary} \\ \\ \mathsf{PREV\_IN}_p^w(b) & \textbf{otherwise} \end{cases}$$

Boundary Blocks

Equations for propagating reaching information across boundary blocks during the first and second phases of forward flow analysis.

$$\mathsf{UIN}_p(boundary\_blk) = \bigcap_{b_j \in pred(boundary\_blk)} \mathsf{UOUT}_p(b_j)$$

$$\mathsf{UOUT}_p(boundary\_blk) = \mathsf{UIN}_p(boundary\_blk) - \biguplus_{p' \neq p} \mathsf{KILL\_IN}^w_{p'}(boundary\_blk)$$

$$\mathsf{CIN}^w_p(boundary\_blk) = \bigcup_{b_j \in pred(boundary\_blk)} \mathsf{COUT}^w_p(b_j)$$

$$\mathsf{COUT}^w_p(boundary\_blk) = \mathsf{CIN}^w_p(boundary\_blk) - \biguplus_{p' \neq p} \mathsf{KILL\_IN}^w_{p'}(boundary\_blk)$$

$$\mathsf{KILL\_IN}^w_p(boundary\_blk) = \bigcup_{b_j \in pred(boundary\_blk)} \mathsf{KILL\_OUT}^w_p(b_j)$$

$$\mathsf{KILL\_OUT}^w_p(boundary\_blk) = \phi$$

$$\mathsf{PREV\_IN}^w_p(boundary\_blk) = \bigcup_{b_j \in pred(boundary\_blk)} \mathsf{PREV\_OUT}^w_p(b_j)$$

$$\mathsf{PREV\_OUT}^w_p(boundary\_blk) = \mathsf{PREV\_IN}^w_p(boundary\_blk) \bigcup \mathsf{KILL\_IN}^w_p(boundary\_blk)$$

Summarizing Intervals

Equations for summarizing reaching information for intervals during the first phase of forward flow analysis. (If b is a basic block, $\mathsf{UREF}_p(b)$ and $\mathsf{CREF}^w_p(b)$ can be computed directly, $\mathsf{KILL\_REF}^w_p(b) = \mathsf{CREF}^w_p(b)$, and $\mathsf{PREV\_REF}^w_p(b) = \phi$.)

Non-Loop Intervals

Assume I is the non-loop interval represented by block $summary\_blk$ and $exit\,(I)$ is the set of exit blocks included in interval I.

$$\mathsf{UREF}_p(summary\_blk) = \bigcap_{b_j \in exit(I)} \mathsf{UOUT}_p(b_j)$$

$$\mathsf{CREF}^w_p(summary\_blk) = \bigcup_{b_j \in exit(I)} \mathsf{COUT}^w_p(b_j)$$

$$\mathsf{KILL\_REF}^w_p(summary\_blk) = \bigcup_{b_j \in exit(I)} \mathsf{KILL\_OUT}^w_p(b_j)$$

$$\mathsf{PREV\_REF}^w_p(summary\_blk) = \bigcup_{b_j \in exit(I)} \mathsf{PREV\_OUT}^w_p(b_j)$$

Loop Intervals

Assume that block *summary_blk* represents a loop interval with exit block *exit_blk*.

$$\mathsf{UREF}_p(summary\_blk) = \begin{cases} \biguplus_{1\le l\le lmax} [\mathsf{UOUT}_{p,l}(exit\_blk) & \textbf{if some execution} \\ \quad -\biguplus_{p'\neq p, l\le l'<lmax} \mathsf{KILL\_OUT}^w_{p',l'}(exit\_blk) & \text{path through} \\ \quad -\biguplus_{p'\neq p, l<l'\le lmax} \mathsf{PREV\_OUT}^w_{p',l'}(exit\_blk) & summary\_blk \text{ crosses} \\ \quad -_R\biguplus_{l<l'\le lmax} \mathsf{COUT}^w_{p,l'}(exit\_blk)\Big] & \text{an epoch boundary} \\ \\ \biguplus_{1\le l\le lmax} [\mathsf{UOUT}_{p,l}(exit\_blk) & \\ \quad -_R\biguplus_{l<l'\le lmax} \mathsf{COUT}^w_{p,l'}(exit\_blk)\Big] & \textbf{otherwise} \end{cases}$$

$$\mathsf{CREF}^w_p(summary\_blk) = \begin{cases} \biguplus_{1\le l\le lmax} \big[\mathsf{COUT}^w_{p,l}(exit\_blk) & \textbf{if some execution} \\ \quad -\biguplus_{p'\neq p, l\le l'<lmax} \mathsf{KILL\_OUT}^w_{p',l'}(exit\_blk) & \text{path through} \\ \quad -\biguplus_{p'\neq p, l<l'\le lmax} \mathsf{PREV\_OUT}^w_{p',l'}(exit\_blk)\big] & summary\_blk \text{ crosses} \\ & \text{an epoch boundary} \\ \\ \biguplus_{1\le l\le lmax} \mathsf{COUT}^w_{p,l}(exit\_blk) & \textbf{otherwise} \end{cases}$$

$$\mathsf{KILL\_REF}^w_p(summary\_blk) = \begin{cases} \mathsf{KILL\_OUT}^w_{p,lmax}(exit\_blk) & \textbf{if every execution} \\ & \text{path through} \\ & summary\_blk \text{ crosses} \\ & \text{an epoch boundary} \\ \\ \biguplus_{1\le l\le lmax} \mathsf{KILL\_OUT}^w_{p,l}(exit\_blk) & \textbf{otherwise} \end{cases}$$

$$\mathsf{PREV\_REF}^w_p(summary\_blk) = \begin{cases} \Big[\biguplus_{1\le l\le lmax} \mathsf{PREV\_OUT}^w_{p,l}(exit\_blk)\Big] & \textbf{if some execution} \\ \quad \bigcup \Big[\biguplus_{1\le l<lmax} \mathsf{KILL\_OUT}^w_{p,l}(exit\_blk)\Big] & \text{path through} \\ & summary\_blk \text{ crosses} \\ & \text{an epoch boundary} \\ \\ \phi & \textbf{otherwise} \end{cases}$$

Unsummarizing Intervals

Equations for processing entry blocks when unsummarizing loops during the second phase of forward flow analysis.

Non-Loop Intervals

Assume that *entry_blk* is the entry block of a non-loop interval summarized by *summary_blk*. ($\mathsf{PREV\_IN}^w$ is used only during the first phase to compute $\mathsf{PREV\_OUT}^w$, so no equation for unsummarizing this set is presented.)

$$\mathsf{UIN}_p(entry\_blk) = \mathsf{UREF}_p(summary\_blk)$$

$$\mathsf{CIN}^w_p(entry\_blk) = \mathsf{CREF}^w_p(summary\_blk)$$

$$\mathsf{KILL\_IN}^w_p(entry\_blk) = \mathsf{KILL\_REF}^w_p(summary\_blk)$$

Loop Intervals

Assume that *entry_blk* is the entry block of a loop interval summarized by *summary_blk*. ($\mathsf{PREV\_IN}^w$ is only used during the first phase. Hence, no equation for unsummarizing this set is presented.)

$$
\mathsf{UIN}_{p,l}(entry\_blk) = \begin{cases}
(\mathsf{UIN}_p(summary\_blk) & \\
\quad - \biguplus_{p' \neq p, 1 \leq l'' < l} \mathsf{PREV\_OUT}^w_{p',l''}(exit\_blk) & \\
\quad - \biguplus_{p' \neq p} \mathsf{KILL\_IN}^w_{p'}(summary\_blk) & \\
\quad - \biguplus_{p' \neq p, 1 \leq l'' < l-1} \mathsf{KILL\_OUT}^w_{p',l''}(exit\_blk) & \\
\quad -_R \biguplus_{1 \leq l'' < l} \mathsf{COUT}^w_{p,l''}(exit\_blk)\big) & \\
\bigcup \Big[\biguplus_{1 \leq l' < l} \big(\mathsf{UOUT}_{p,l'}(exit\_blk) & \textbf{if } l > 1 \textbf{ and} \\
\quad - \biguplus_{p' \neq p, l' < l'' < l} \mathsf{PREV\_OUT}^w_{p',l''}(exit\_blk) & \text{some execution path} \\
\quad - \biguplus_{p' \neq p, l' \leq l'' < l-1} \mathsf{KILL\_OUT}^w_{p',l''}(exit\_blk) & \text{through } summary\_blk \\
\quad -_R \biguplus_{l' < l'' < l} \mathsf{COUT}^w_{p,l''}(exit\_blk)\big)\Big] & \text{crosses an epoch boundary} \\
 & \\
(\mathsf{UIN}_p(summary\_blk) & \\
\quad -_R \biguplus_{1 \leq l'' < l} \mathsf{COUT}^w_{p,l''}(exit\_blk)\big) & \\
\bigcup \Big[\biguplus_{1 \leq l' < l} \big(\mathsf{UOUT}_{p,l'}(exit\_blk) & \\
\quad -_R \biguplus_{l' < l'' < l} \mathsf{COUT}^w_{p,l''}(exit\_blk)\big)\Big] & \textbf{otherwise}
\end{cases}
$$

$$
\mathsf{CIN}^w_{p,l}(entry\_blk) = \begin{cases}
(\mathsf{CIN}^w_p(summary\_blk) & \\
\quad - \biguplus_{p' \neq p, 1 \leq l'' < l} \mathsf{PREV\_OUT}^w_{p',l''}(exit\_blk) & \\
\quad - \biguplus_{p' \neq p} \mathsf{KILL\_IN}^w_{p'}(summary\_blk) & \\
\quad - \biguplus_{p' \neq p, 1 \leq l'' < l-1} \mathsf{KILL\_OUT}^w_{p',l''}(exit\_blk)\big) & \textbf{if } l > 1 \textbf{ and} \\
\bigcup \Big[\biguplus_{1 \leq l' < l} \big(\mathsf{COUT}^w_{p,l'}(exit\_blk) & \text{some execution path} \\
\quad - \biguplus_{p' \neq p, l' < l'' < l} \mathsf{PREV\_OUT}^w_{p',l''}(exit\_blk) & \text{through } summary\_blk \\
\quad - \biguplus_{p' \neq p, l' \leq l'' < l-1} \mathsf{KILL\_OUT}^w_{p',l''}(exit\_blk)\big)\Big] & \text{crosses an epoch boundary} \\
 & \\
\mathsf{CIN}^w_p(summary\_blk) & \\
\bigcup \big(\biguplus_{1 \leq l' < l} \mathsf{UOUT}_{p,l'}(exit\_blk)\big) & \textbf{otherwise}
\end{cases}
$$

$$
\mathsf{KILL\_IN}^w_{p,l}(entry\_blk) = \begin{cases}
\mathsf{KILL\_OUT}^w_{p,l-1}(exit\_blk) & \textbf{if } l > 1 \textbf{ and every} \\
 & \text{execution path} \\
 & \text{through } summary\_blk \\
 & \text{crosses an epoch boundary} \\
 & \\
\big(\biguplus_{1 \leq l' < l} \mathsf{KILL\_OUT}^w_{p,l'}(exit\_blk)\big) & \\
\bigcup \mathsf{KILL\_IN}^w_p(summary\_blk) & \textbf{otherwise}
\end{cases}
$$