Abstract

Reduction variables are an important class of cross-thread dependence that can be parallelized by exploiting the associativity and commutativity of their operation. In this paper, we define a class of shared variables called partial reduction variables (PRV). These variables either cannot be proven to be reductions or they violate the requirements of a reduction variable in some way.

We describe an algorithm that allows the compiler to detect PRVs, and we also discuss the necessary requirements to parallelize detected PRVs. Based on these requirements, we propose an implementation in a TLS system to parallelize PRVs that works by a combination of techniques at compile time and in the hardware. The compiler transforms the variable under the assumption that the reduction-like behavior proven statically will hold true at runtime. However, if a thread reads or updates the shared variable as a result of an alias or unlikely control path, a lightweight hardware mechanism will detect the access and synchronize it to ensure correct execution. We implement our compiler analysis and transformation in GCC, and analyze its potential on the SPEC CPU 2000 benchmarks. We find that supporting PRVs provides up to 46% performance gain over a highly optimized TLS system and on average 10.7% performance improvement.

Categories and Subject Descriptors C.1.4 [Processor Architectures]: Parallel Architectures; D.1.3 [Programming Techniques]: Concurrent Programming; D.3.4 [Processors]: Compilers

General Terms Algorithms, Design, Measurement, Performance

Keywords Reduction variables, parallelization, Thread-Level Speculation, multi-core architecture

1. Introduction

Given the abundance of multi-core architectures and the relatively few programs capable of exploiting parallel architectures effectively, techniques for automatic and semi-automatic parallelization are increasingly important. Parallelizing compilers attempt to decompose a program into threads (or tasks) that can execute in parallel when there are no cross-task control or data dependences. Despite their many advances, such compilers still fail to parallelize many codes. Examples of such codes are accesses through pointers, subscripted subscripts, interprocedural dependences or input dependent access patterns. Parallelizing compilers [16, 20, 26, 39] that target architectures with support for dynamic speculation of data dependences [13, 15, 24, 25, 29, 30, 40] have shown promise for coping with these kinds of problems. However, to move beyond the limitations of classical parallelizing optimizations, new parallelizing compilers must look to analyses and transformations that do not rely on conservative assumptions for correctness, but rather consider new ways to speculate on the likely behavior of data and control dependences at runtime. We apply this observation in the context of reduction variables.

Reduction variables are an important class of loop-carried dependences in programs. They are characterized by an expression of the following form inside a loop: \( r \leftarrow r \circ expression \), where \( r \) is the reduction variable and \( expression \) is a value computed independently form \( r \). \( \circ \) is assumed to be commutative and associative. Furthermore, \( r \) is not used or defined anywhere else in the loop. Even though the computation results in a loop-carried (and likely cross-task) dependence, it can still be parallelized by exploiting the commutative and associative properties of the operation to avoid a cross-task dependence. Each thread can accumulate to lists, requiring prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © 2010 ACM 978-1-60558-635-9/10/04…$10.00

• Defines a new class of variables called partial reduction variables and characterizes their presence in SPEC CPU 2000 applications.
• Describes an algorithm to detect PRVs automatically and validate manually marked PRVs to promote programmability.
• Identifies the needed hardware and software supports for PRV parallelization.
• Evaluates the impact of PRV parallelization with our proposed architectural support on SPEC CPU 2000 applications. On a set of SPEC CPU 2000 applications, we found that supporting
PRVs provides up to 46% performance gain over a highly opti-
mized TLS system and on average a 10.7% improvement.

The remainder of the article is organized as follows: Section 2 defines PRV and characterizes their occurrence in SPEC CPU 2000 applications; Section 3 describes the compiler algorithm for detect-
ing PRVs, and Section 4 describes how to parallelize them; Section 5 discusses the architectural support added to enable PRV parallelization; Section 6 describes our experimental setup and presents result; the final section concludes.

```plaintext
for( ... ) { 
    for( ... ) { 
        // alias with delta_vert_cost 
        *costptr += ABS( newx - new_mean ) ... ; 
    } 
    ... 
    // tmp_rows is global array 
    rowsptr = tmp_rows[net] ; 
    // rowsptr aliases with delta_vert_cost and *costptr 
    2--------> for( row = 0 ; rowsptr[row] == 0 ; row++ ) ; 
    ... 
    // alias with *costptr and delta_vert_cost 
    3--------> tmp_missing_rows[net] = -m ; 
    // alias with *costptr 
    4--------> delta_vert_cost += ... } 
```

Figure 1. The new_dbox_a function taken from 300.twolf.

2. Partial Reduction Variables

A reduction is kind of loop recurrence that can be parallelized on a multi-core (or multiprocessor) system easily using conventional hardware. A reduction variable has the following form inside a loop: $r = r \odot \text{expression}$. Here, $\odot$ is a suitable reduction op-
eration, and expression is independent of $r$. There are additional constraints placed on $r$. It cannot be read or written in any other statement in the loop. With these restrictions in place, the compu-
tation of $r$ can be parallelized by exploiting the associativity and commutativity of the $\odot$ to group expressions so that there are no cross thread dependences.

Reduction variables are an important case to handle when paral-
lelizing code, even irregular codes like SPEC CPU [19, 39]. How-
ever, the restrictions placed on a reduction variable can be pro-
hibitive in integer or other highly irregular codes. In particular, the requirement that an RV cannot be read or written outside of the re-
duction operation prevents the inclusion of many variables. There are many reasons that prevent inclusion. First, conservative pointer analysis results in the appearance of possible reads or writes outside of the reduction statement. Second, cross-module or library calls that cannot be fully analyzed is part of 300.twolf. This means that reads or writes of the RV can occur that may escape to that call. Finally, rarely executed code paths result in a known read or write outside of the re-
duction statement that will usually not occur at runtime. While the compiler is not able to parallelize these cases, they may behave like reductions at runtime. Instead of letting these opportunities pass by, we want to take advantage of the associativity and commutativity of these operations. Since they do not conform to strict definition of a reduction variable, we call them Partial Reduction Variables (PRVs).

Part of our inspiration for pursuing PRVs came from this loop in the new_dbox_a function in dbox.c which is part of 300.twolf in the SPEC CPU 2000 suite. There are two variables (labeled 1 and 4) that have clear reduction patterns in the loop shown in Figure 1 (N.B. some code has been omitted). It would appear from inspection that they are ideal candidates for traditional RV optimization. There appear to be no other reads or writes to the variables in the loop, and the operation type is addition. However, aliases are present in this loop (shown at 2 and 3). The aliasing write at 3 may be a definition of one of the potential RVs, and the aliasing read at 2 may use one of the potential RVs. The presence of either one is enough to disqualify both variables. However, even if 2 and 3 are removed, the problem remains since 1 and 4 may alias each other. To preserve correctness of the program at compile time, neither variable can be labeled as an RV since an alias may occur among the two RVs or with other variables in the loop.

In the remainder of this section, we will provide a definition of PRV and will characterize the frequency of PRVs in the SPEC CPU 2000 applications.

2.1 Definition of Partial Reduction Variables

To encompass the behavior shown in twolf we will relax some of the restrictions on an RV to create a formal definition for Partial Reduction Variable. However, first, we specify the definition of RV we will use for the remainder of this section.

A variety of techniques have been explored for detecting redu-
duction variables [1, 4, 12, 17]. Our approach builds on the body of work that searches for specific patterns in the data dependence graph (DDG) that indicate a reduction variable. A reduction oper-
ation forms a cycle in the DDG that begins with the load of the reduction variable (RV-load), includes a series of true dependences that perform the operation, and updates the reduction variable (RV-
store). The cycle is complete because the last store feeds back to the first load along a back-edge in the control flow graph (CFG). For convenience, we will call this an RV-cycle. For the computed RV-cycle, we identify a sub-graph from the RV-load to the RV-store, called the RV-update-chain. In order to detect a qualified RV-cycle, a few other properties are enforced: (1) none of the intermediate computations in the cycle can propagate to other variables, (2) there are no reads or writes of the reduction variable outside of the cycle, and (3) the chain of operations in the cycle form an associative and commutative operation. Note that this definition allows multiple RV-update-chains in the RV-cycle as long as they all must update the same variable using the same reduction operator. In addition, we tolerate RV-update-chains on multiple paths (in a sub-loop or in both an if-then and if-else statement) as long as all paths through the RV-cycle form a sequence of valid updates.

For PRVs we start with the same definition as RV, but we re-
lax two properties, (1) and (2), of our previous definition. First, we allow both explicit and implicit references to the PRV outside of the RV-cycle. This means that reads or writes of the PRV are allowed that are inconsistent with the reduction operation. These reads and/or writes may come from any of the sources described earlier: aliased memory references, un-analyzed control paths that reference the PRV, or known references on analyzed paths. Even though the reads or writes may be implicit or explicit, we will refer to them all as a PRV-may-ref. Second, we relax the requirement that all paths through the RV-cycle form a sequence of valid up-
dates. We must relax this property in order to allow a PRV-may-ref outside of an RV-update-chain. However, it also gives us consid-
erable freedom when labeling PRVs. Instead of restricting the static control flow patterns, our definition is now flexible enough to iden-
tify reductions that may occur at runtime.

We will place some restrictions on where a PRV-may-ref oc-
curs in the code. The RV-update-chain covers the full set of paths that begin with an RV-load and end with an RV-store. We allow a PRV-may-ref along any path incident with RV-cycle except for those in the RV-update-chain. These restrictions are not arbitrary, rather they are introduced to reduce the complexity of our proposed support or to improve performance. The full rationale for these re-
strictions may not be apparent until we describe our entire system.

Figure 2 presents several examples of our definition. In all of the charts in this figure, the outlined box represents a loop, $r_n$ is
A 1 = Φ(A0, A 7 )
A 4 = Φ(A3, A 2 )
A 6 = Φ(A5,A 1 )
t 4 = *a <A6 >
t 5 = t 4 + ...<A 7 > *a = t 5 ...
while(...)
 *a += ...
} else {
 ... = *a
 *a += ...
} while (...)  
(a) (b)

Figure 2. PRV Regions for privatization.

There is considerable variation of both RV and PRV presence in these applications. Interestingly, our definition allowed us to classify significantly more variables as possible reductions than before, including the ones in twolf that inspired our definition. Based on the average number of RVs and PRVs, we increased the number of reduction variables considered for parallelization by a factor of 3.

3. PRV Detection

To effectively exploit PRVs in integer and irregular codes, we must first detect them. In this section, we will describe our algorithm for automatically detecting PRVs on an intermediate representation of the program. However, because our automatic detection pass occurs after many other optimizations, we also describe support for manually marking PRVs using pragmas inserted by the heroic programmer. The pragma is a hint not a mandate, and PRVs marked using the pragma must be validated by our automatic selection pass. We will describe both features, and how they work together to detect PRVs.

<table>
<thead>
<tr>
<th>Application</th>
<th>RV</th>
<th>PRV</th>
</tr>
</thead>
<tbody>
<tr>
<td>crafty</td>
<td>26</td>
<td>6</td>
</tr>
<tr>
<td>gap</td>
<td>48</td>
<td>88</td>
</tr>
<tr>
<td>gzip</td>
<td>9</td>
<td>27</td>
</tr>
<tr>
<td>mct</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>parser</td>
<td>28</td>
<td>12</td>
</tr>
<tr>
<td>perlbmk</td>
<td>14</td>
<td>28</td>
</tr>
<tr>
<td>twolf</td>
<td>45</td>
<td>56</td>
</tr>
<tr>
<td>ammp</td>
<td>8</td>
<td>17</td>
</tr>
<tr>
<td>art</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>equake</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>mesa</td>
<td>5</td>
<td>156</td>
</tr>
<tr>
<td>Average</td>
<td>18</td>
<td>38</td>
</tr>
</tbody>
</table>

Table 1. Frequency of PRVs and RVs in SPEC CPU 2000 subset.

We implemented our definitions of RV and PRV in an analysis pass in GCC 4.3 [11] to assess the frequency of these variables in SPEC CPU 2000 applications. Table 1 shows the number of RVs and PRVs found for each application we analyzed. This is not an upper bound on the number of PRVs, but it is the number detected by our algorithm (Section 3). The second column shows the number of RVs and the third column shows the number of additional PRVs that are detected. Note that the columns are disjoint sets even though a PRV is a superset of RV.

3.1 PRV Detection Algorithm

Our PRV detection algorithm is implemented in GCC 4.3 using the Tree-SSA intermediate representation [8, 11]. This is an SSA representation that additionally annotates the SSA web with points-to set information. Each separate points-to set is given a symbolic name, and the definitions and uses of that set are converted into valid symbolic SSA form. Figure 3 illustrates such annotation. Fig.3(a) shows the original code, and Fig.3(b) shows the corresponding
CFG in SSA form with the memory name \( A \) representing a points-to set. References to \( A \) in angled brackets represent a definition or use of \( A \) appropriate.

The PRV detection algorithm looks for an RV-cycle. As described in Section 2, the RV-cycle is a circuit of update operations on a single variable using a single reduction operator. The example in Figure 3 has such circuit of updates if you follow the edges from \( 1 \rightarrow 2 \rightarrow (6,7) \rightarrow 4 \rightarrow 5 \). There is a use in block 8 and a definition in block 3 but these do not prevent classification as a PRV.

Since detecting recurrences on the CFG is well understood [1, 21], the algorithm will be described briefly. For each phi-node in a loop's header block, we will search to see if it's part of an RV-cycle. Figure 3 shows one such \( \phi \) in the loop header block in block 1. We recursively walk backward along the use-def chains of the SSA graph. Therefore, to begin our search, we start with the statement that defines the \( \phi \)'s argument along the backedge of the loop (definition \( < A \rightarrow \)). From here, there are a few possibilities that we must handle.

**Assignment statement.** If it is an assignment to the reference we are searching for, then from this statement we search for an RV-update-chain. Detecting the update chain involves following the dependence chains of the assignment backward until finding a PRV load. During the traversal, all dependences are searched along use-def chains until they either reach the PRV load, a constant expression, a non-aliasing memory load, or exit the basic block. On the return path after reaching a terminating condition, the algorithm collects details about the operation. By merging the results from all operands in an expression and taking the expression operators into account, we can ensure the validity of each update operation. Considering block 5 in Figure 3, it is clear that tracking back along the use-def chain will yield a valid chain quickly: \( A7 \rightarrow t5 \rightarrow t4 \rightarrow A6 \). After finding a chain, we continue searching backward for a complete cycle from the load operation.

If we fail to validate an RV-update-chain or determine the statement to be the definition of a possible alias (as is the case in block 3), we mark the store as a non-updating store and traverse back to its prior dominating definition (the one it kills) and look for an RV-update-chain starting there. This step is important since it allows us to overlook definitions caused by aliasing writes, un-analyzed functions, or even explicit writes that are not part of an update.

**Conditional \( \phi \).** If we find a conditional \( \phi \), we branch back along both paths recursively and continue searching for a statement that writes to the PRV. In order for the recursive call to return in the affirmative, one of the paths followed backward must include a valid RV-update-chain. From the example, the \( \phi \)-node in block 4 would be the next statement encountered. This is a condition-\( \phi \), and so we search backward along both paths. Since the path through the \( 6 \rightarrow 7 \) inner-loop includes an RV-update-chain, the eventual result of analyzing this statement is that a valid RV-update-chain was found. If RV-update-chain were found along both paths, then the type of reduction operations must match.

**Inner-loop \( \phi \).** If an inner-loop \( \phi \) is encountered, the entire detection algorithm is called recursively on the inner loop and analyzed for the particular PRV of interest. In the example, we would apply the same detection mechanism described thus far, but restricted to the inner-loop \( 6 \rightarrow 7 \). Inside the loop, the RV-update-chain \( A3 \rightarrow t1 \rightarrow t0 \rightarrow A2 \) is identified. In addition, the input argument to the inner-loop \( \phi \) that is not from a back edge which is also searched recursively. If an RV-update-chain is found both in the inner-loop and along the backward use-def chain, then their operation types must match.

**Header \( \phi \) or outside region.** The statement found could be the header phi we started from. This will not happen initially, but must eventually occur to terminate the cycle. If the header \( \phi \) or a statement outside the loop is reached, the search terminates. As long as one path through the loop back to the header includes an RV-update-chain, we mark it as a potential PRV.

**Final Validation.** As the final step, we must verify that no PRV-may-reference occurs in any RV-update-chain in the RV-cycle. For each RV-update-chain detected during traversal, we iterate from the use to the definition and search for explicit or implicit (aliasing) references to the PRV in between. In addition, we also guarantee that each definition in the RV-update-chain is only consumed within the chain. This is trivially computed using the SSA graph.

### 3.2 Manual Detection & Auto-Validation

Manual marking of potential PRVs is done using pragmas added directly to the source code. Pragmas allow a heroic programmer the opportunity to suggest good PRVs. We found this support important in order to enable effective task selection for our automatic pass. Automatic parallelization environments often use many heuristics to decide on effective decompositions. With effective hints, these compilers can work better.

The syntax of the pragmas is very similar to OpenMP's reduction annotation. However, pragmas in our system are interpreted as a strong hint but not as a mandate. To validate PRVs marked using pragmas, we run the detection pass, as described in the previous section, on the marked loop and variable name. If it does not meet the definition, it is discarded. This enables even not-so-heroic programmers to suggest PRVs without breaking our system.

A key challenge in making this mechanism work is preserving the PRV until the PRV analysis pass runs. The pragma specifies the PRV using its lexical name, but these names are often lost after lowering to an intermediate representation and optimizations have been performed. To ensure that a manually marked PRV remains visible to our analysis pass, we replaced each static reference of a specified PRV (e.g. `realPRV`) with a volatile reference of equivalent type (e.g. `volatilePRV`) since volatile variables are ignored by compiler optimizations that may eliminate, move, or rename memory references. We inserted assignments of `volatilePRV = realPRV;` before and `realPRV = volatilePRV;` after the pragma-marked region.

### 4. Speculative Parallelization of PRVs

In this section, we discuss the parallelization strategy for PRVs and the kinds of systems that can support it.

#### 4.1 Parallelization Requirements

PRV parallelization involves lowering the RV-update-chains found in the loop to run efficiently in parallel. If the PRVs behave only like classical RVs then their parallelization can be highly efficient, and the strategies used can be similar to those that support RVs. Each reference to the PRV in an RV-update-chain is replaced with a reference to private variable of the same type. Before executing the parallel region, the private is initialized to a neutral value suitable for the RV operation. Either during or after the parallelized region, the private is accumulated with the PRV. Loop unrolling can be used to lengthen the region and increase the gains achieved from parallelization.

However, in the event that loads and stores to the PRV do occur outside of the RV-update-chains, the first requirement is that they have to be detected. Detection is not trivial since the compiler does not mark each PRV access. For explicit reads or writes, the compiler should handle them directly. For aliases analyzed by the compiler, runtime disambiguation tests can be used. For accesses in un-analyzed control paths, the parallelization environment must detect the access to the PRV.

Upon detection of PRV accesses outside an RV-update-chain, additional support is needed to ensure correct parallelization. The possible behaviors of PRVs can be divided into three cases based on the types of PRV reference that may occur: stores only, loads only, and a combination of loads and stores.

**Stores only.** In the case that only stores to the PRV may occur outside of the RV-update-chain, then care must be taken to ensure that the value of the PRV after executing the loop is equal to the last
non-updating store accumulated with the private variable updates from all later iterations. This requires preserving the last store and ordering it with respect to all later iterations (and accumulations in the same thread). Fortunately, this does not require synchronization with other threads.

**Loads only.** In the case that only loads to the PRV may occur outside of the RV-update-chain, the potential overhead is much higher. When the load occurs, all private values accumulated from prior iterations must be merged with the PRV to provide the correct value. This requires that all prior iterations complete their last update to the private variable. If some of those iterations have not yet been scheduled, then the load must wait (or speculate on the loaded value).

**Loads and stores.** In the event that both a load and store may occur, then a combination of corrective actions are needed for both the load or store. Furthermore, the mechanisms must cooperate because a load must get the accumulated result since the last store. The overhead and efficiency of the mechanism can vary depending on the actual pattern of loads and stores in the loop. If stores frequently precede loads (especially in the same iteration), then the loads need not wait long to accumulate an up-to-date value. However, if loads and stores are far apart, then performance may be appear more like the loads only case.

### 4.2 System for PRV Parallelization

Aspects of PRV parallelization are currently supported by some systems, but none fully support it. If PRV selection is restricted to the case of only stores occurring outside of the RV-update-chain, then a system like Thread-Level Speculation (TLS) [15, 24, 25, 29, 30] could support it with few modifications. The latest store to the PRV would overwrite all previous stores. When the accumulation is done after the parallel region (or as part of each iteration), the most recent store would be used. However, somehow, the accumulations from iterations prior to the store still must be discarded. Transactional Memory [14, 33] (TM) could work similarly if a partial or-dering were imposed on tasks that wrote to the PRV. However, these systems cannot fully support the case of loads or stores by default.

To fully support our definition of PRV, we extend a system that implements Thread-Level Speculation to record the link between the private variable and the PRV. By exposing this link to the TLS system, we can properly correct loads and optimize the handling of stores. Since TLS naturally supports thread ordering, managing the corrective actions for all task behaviors is straightforward.

#### 4.2.1 Thread-Level Speculation

In this section, we briefly review the key concepts of Thread-Level Speculation. A TLS compiler breaks a hard-to-analyze sequential code into tasks, and speculatively executes them in parallel, hoping not to violate sequential semantics (e.g. [2, 5, 31, 32, 36]). The control flow of the sequential code imposes a control dependence relation between the tasks. This relation establishes a total order among the tasks, and we can use the terms predecessor and successor to express this order. This ordering also determines a data dependence relation on the memory accesses issued by the different tasks that parallel execution cannot violate.

A task is **speculative** when it may perform or may have performed operations that violate data or control dependences with its predecessor tasks. When a non-speculative task finishes execution, it is ready to **commit**. The role of commit is to inform the rest of the system that the data generated by the task are now part of the safe, non-speculative program state. Among other operations, committing always involves passing the non-speculative status to a successor task. Tasks must commit in strict order from predecessor to successor. If a task reaches its end and is still speculative, it cannot commit until it acquires non-speculative status.

As tasks execute in parallel, the system must identify any violations of cross-task data dependences. Typically, this is done with special hardware support that tracks, for each individual task, the data written and the data read without first writing it. A data dependence violation is flagged when a task modifies a version of a datum that may have been loaded earlier by a successor task. At this point, the consumer task is **squashed** and all the state that it has produced is discarded. Its successor tasks are also squashed. Then, the task is re-executed.

TLS architectures can discard the state produced by a task and re-start the task thanks to special hardware that buffers all speculative modifications, and a checkpointing mechanism that enables rollback. Note that anti and output dependences across tasks do not cause squashes.

#### 4.2.2 Example

Figure 4 explains how we support PRV parallelization on a TLS-based system. Fig.4(a) shows the original code. The TLS compiler through heuristics or programmer hints chooses tasks for parallelization. Note, that the TLS compiler can overlap any dependence due to hardware that will catch any dependence violation at runtime. The pragma at 1 in Fig.4(a) suggests to the compiler a potential good TLS task (the region marked in between 1 and 1'), which includes a PRV named *sum* whose reduction operator is \(+\).

At location 2, *sum* operates as a traditional reduction variable, but it is read and written at locations 3 and 4 respectively, which make it a PRV.

Fig.4(b) shows the transformed codes needed to create parallel tasks in TLS and support for the PRV. The `spawn()` at 1 will create a new thread beginning after the `commit()` at 1'. Similar to how a reduction is handled traditionally, *sum* is privatized as `priv` (as 2, 2', and 2" show). But in order to correctly support the load at 3 and store at 4, additional actions are inserted as 3' and 4'. In the case of the load, the first action is `become_safe()`. This guarantees the
task will wait for the final version of sum from its predecessor task before reading it. Next, it updates sum with its accumulated value in priv and clears priv. After all this is done, it can use the value in sum. Also, because we cleared priv, additional accumulation into priv that could occur within the task will work correctly. In the case of the store at 4, all we need to do is clear priv at 4\textsuperscript{th}. At the end of the task, just before commit, priv is accumulated into sum. To ensure this does not happen prematurely, we synchronize the update using become\_safe().

If the compiler can fully analyze sum statically and make sure it has no alias, no hardware support is needed and Fig\textsuperscript{6}(b) shows the transformed code added by the compiler. Otherwise, if the accesses at 3 and 4 were in fact aliases, then the instructions are not inserted by the compiler, instead hardware should detect the access to sum and take the same set of actions. This requires the hardware knowing the link between priv and sum and interpreting the read or write as occurring outside of the RV-update-chain.

Fig\textsuperscript{4}(c) shows how program correctness is guaranteed at runtime. Time proceeds down the page while tasks to the right are more speculative. Iteration N+1 contains the store at 4. The block to the right of label 4 shows the correcting actions. Note that it stalls until it can load the final accumulated value of sum from its predecessor. Iteration N+2 contains the load at 3 and its correcting actions. The synchronization actions needed to bring sum up to date do lead to stalls, but these effects are ameliorated by a couple of factors. We expect these synchronization events to be infrequent in PRV tasks selected by the TLS compiler, and even when they do occur, it is better to synchronize than pay the price of a squash.

5. Architectural Support for PRVs

We add new architectural features to a TLS system to support PRV parallelization. Figure 5 shows the key pieces of our new architecture: (1) the PRV Lookup Table (PLUT) tracks the mapping from PRV to private variable and provides key details for synchronizing updates, (2) the PRV Signature stores a hash of the addresses of all the PRV’s for quick access, and the PRV Controller which implements the detection, correction, and update synchronization algorithm. The Load-Store Queue (LSQ) and Versioned Cache are assumed to be typical with no features specific to our scheme.

![](image)

Figure 5. Architectural modifications to enable PRV parallelization.

5.1 Support for a PRV Access

If a PRV has or may have aliases, the compiler alone cannot totally handle it. Detecting and correcting such PRV accesses occurs at runtime when a reduction operation is in progress. Before entering a region of code with such a PRV the compiler schedules a pair\_addr instruction to notify the hardware that a partial reduction operation is under way and links the address of the PRV to the private variable holding the partial state. Hardware keeps a record of the PRV and private variable in the PRV Lookup Table. The hardware must assume that the reduction is underway until it receives an unpair\_addr instruction. During this window of execution, hardware monitors for illegal accesses to the PRV. It is the compiler’s responsibility to insert pair\_addr and unpair\_addr and ensure no illegal accesses occur to the PRV outside the region. Table 2 gives the descriptions of pair\_addr and unpair\_addr instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pair_addr op t Rv, Rp</td>
<td>Pair reduction variable, whose address is in Rv, with the private variable whose address is in Rp. Op describes the operator (addition, multiplication), and t indicates whether the PRV is an int, float, double, or long long for proper initialization.</td>
</tr>
<tr>
<td>unpair_addr Rv</td>
<td>Unpair the reduction variable from any private variable.</td>
</tr>
</tbody>
</table>

When a pair\_addr instruction is executed for the first time on a PRV, a new entry is created in the PRV Lookup Table, shown in Figure 6. The pair\_addr instruction allocates an entry in the table, sets the Valid (V) bit, clears the Unpaired (U) bit (described later), sets the Private variable address, and initializes Accumulated Partial Result. The PRV address is also added to the PRV Signature. The signature is similar to a Bloom filter [3] and provides a summary of all the addresses in the lookup table. Searching a signature is much faster than searching a modest size lookup table, so this allows the lookup to be placed off the critical path.

The unpair\_addr instruction indicates that the pairing of PRV to the private location should be stopped, and sets the U bit in the corresponding entry indicating that the partial address is no longer paired with the PRV. Also, the value currently stored in the private variable is loaded from memory and accumulated with the Accum field in the PLUT entry. Even though the PRV’s address is marked as unpaired, the entry for the PRV need not be removed from the PRV Lookup Table immediately. Instead, we allow it to continue monitoring this address since the partial state is preserved in Accum. This policy optimistically delays the memory update until the end of the task when it is safe to do so without causing a squash.

5.1.1 Detecting a Conflict

If there is at least one valid entry in the PRV Lookup Table, all loads and stores will be checked against the signature for a conflict. If no conflict is found with the signature, then the table need not be checked. However, if a conflict with the signature is found, then the PRV Lookup Table is searched for a matching PRV. Upon finding a matching entry in the table, the LSQ is temporarily stalled so that the state of the PRV can be updated as necessary. If no match is found, then no action is taken and the cache access occurs as usual.

5.1.2 Correcting the State

When a conflict on a PRV is detected, the PRV Controller will take the necessary actions to fix the program’s state. These actions mirror the code inserted by the compiler when it finds a non-PRV access. We will explain correcting a load and a store separately, but note that the mechanisms also suffice for the case when both loads and stores occur.
Correcting a Load. In the case of load access to the PRV, the Controller will request the current value stored in the PRV. If the Unpaired bit is empty, it also requests the private variable. Note these requests are equivalent to memory requests made by the processor and are handled in exactly the same way as standard TLS coherence request.

Then, it will accumulate, as determined by the operation kind stored in the PLUT entry, the PRV and the Accum field in the PLUT entry. Finally, it will store the result into the PRV’s location in memory and reset the private variable to the correct initial value, depending on the type of variable and reduction operator. At the end of this sequence of operations, the PRV is fully up-to-date and the private variable is reset. Now the LSQ is un-stalled and allowed to complete its load operation on the PRV. When it loads, it will find the correct value in the cache.

Finally, if after correcting a load we discover that the U-bit is set, it is safe to clear the valid bit since the entry no longer contains any partial state. The only reason the entry was still in the PLUT was to delay the PRV update as long as possible. But since the update has occurred, no reason exists to keep the entry.

Correcting a Store. The case of a store access to the PRV is somewhat simpler. In this case, we are simply updating the current value of the PRV, and we could choose not to stall that update. However, we do stall the LSQ momentarily and insert a store to the private variable to return it to its initial value, and we reset the Accum field in the PLUT entry.

Multiple PLUT entries for the same PRV. The compiler may aggressively select two PRVs in the same loop that may alias. Most of the time, they will not access the same location. However, if two entries in the PLUT are added with the same PRV and a different private location, the hardware forces both into a special mode of operation, indicated by the A bit in the PLUT. Instead of monitoring for accesses to the PRV, we monitor accesses of both private variables. If there is a read to the private variable, such read is automatically replaced with a PRV read. Similarly, a write to the private is treated as a write to the PRV. This will guarantee that the PRV is updated correctly with respect to both PRV chains.

5.2 Synchronizing PRV Updates

With the support already described, synchronizing updates is relatively straightforward. When a task has become safe, finished executing all its instructions, and is ready to commit, it scans its PLUT for entries that have its Task Id (TID) and that are valid. For each such entry, it is handled by using the same logic as for correcting a load. This takes care of all actions necessary to bring the PRV up-to-date.

5.3 Compiler Support

This hardware mechanism requires that the PLUT never overflow, lest a mapping between PRV and private variable be lost. To avoid unnecessary complexity in the hardware, we place two restrictions on our technique: (i) we do not allow a PRV task to be nested within another PRV task, and (ii) we require that the compiler manage tappable usage to prevent overflow. In the first requirement, disabling PRV-task nesting ensures that the compiler can determine the number of active PRVs in the PLUT through global analysis of a single subroutine. However, this does complicate task selection because this property must be enforced conservatively – even if such a nesting was unlikely to occur dynamically, if it is ever possible, such a nested task cannot be selected. Such a limitation requires selecting among alternatives, and this selection is carried out explicitly by our profiler (see Section 6).

The second requirement ensures that a task will never use more entries than available in the PLUT, and this is enforced directly through analysis of each task region. We use a compiler parameter NumPRV (the maximum static number of PRVs per task) to limit the number of PRVs per task. Our empirical evidence suggests that up to 4 static PRVs finally survive for an application. So we set NumPRV to 4 and use 4 PLUT entries for each core in the evaluation and found this to be sufficient.

5.4 Cost Estimation

The estimated per-core hardware cost added to support PRVs for a 64-bit system includes a 4-entry PLUT (4 * 34-Byte = 136 Bytes); 1 32-bit signature (Bloom filter with single hash function derived from lower bits of address excluding the two least significant); other control logics needed to control PLUT and utilize existing ALU/FPU during accumulation; and other logics added to support the 3 new instructions (pair_addr, unpair_addr, and become_safe)

The main performance cost added for a store or a non-conflicting load is the delay of a 32-bit signature which is negligible since it can be done in parallel with other logic. However, the cost for a conflicting load is the stalled cycles waiting for the correct value being fed from its predecessor thread (this delay is comparable to that of a synchronization operation), plus the cycles to accumulate this value to that of the privatized variable. This may seem significant, but it is worth it compared to the squash-and-restart cycles that would otherwise occur.

6. Evaluation

We implement our system on a cycle-accurate execution-driven simulator [23]. The simulator models superscalar processors and memory subsystems in detail. The TLS architecture modeled is shown in Table 3. It is a four-processor CMP with TLS support. Each processor is a 3-issue core and has a private L1 cache that buffers the speculative data. The L1 caches are connected through a crossbar to an on-chip shared L2 cache. All communication between cores occurs through the cache coherence protocol.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>1 GHz</th>
<th>ROB</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch width</td>
<td>8</td>
<td>1-window</td>
<td>68</td>
</tr>
<tr>
<td>Issue width</td>
<td>3</td>
<td>LDST queue</td>
<td>49/42</td>
</tr>
<tr>
<td>Retire width</td>
<td>3</td>
<td>Mem/FPU</td>
<td>100</td>
</tr>
<tr>
<td>Branch predictor:</td>
<td></td>
<td>Spawn Overhead</td>
<td>12 cycles</td>
</tr>
<tr>
<td>Mispred. Penalty</td>
<td>14 cycles</td>
<td>Squash Overhead</td>
<td>20 cycles</td>
</tr>
<tr>
<td>BTB</td>
<td>2K, 2-way</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Core details. Cycle counts are in processor cycles.

We implement our compiler pass in a copy of the POSH [16] compiler, which has been ported to GCC 4.3 and is one of the state-of-the-art TLS compilers. We use the compiler to generate 3 different binaries for each application evaluated, shown in Table 4. The Base case is a sequential binary that we normalize all of our plots against. TLS shows the result of the POSH compiler with all of its optimizations enabled. TLS+PRV includes tasks automatically selected using all POSH’s optimizations plus support PRV detection and parallelization. Even though a PRV may be detected and parallelized, POSH leverages a profiling stage to weed out ineffective tasks. The major criteria to evaluate a task includes its size, static and dynamic hoist distance, violation rates, prefetching effect, and costs to parallelize it. Only high quality tasks will survive to the final output.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>-O2</td>
</tr>
<tr>
<td>TLS</td>
<td>POSH optimizations + Base</td>
</tr>
<tr>
<td>TLS+PRV</td>
<td>All POSH tasks plus PRV tasks that survive optimization</td>
</tr>
</tbody>
</table>

Table 4. Compile settings for each application evaluated.
We evaluate our proposal on a set of SPEC CPU 2000 applications. We exclude vortex and econ because the version of the source code we use cannot be compiled. We exclude gcc and perlbench because the TLS infrastructure we use does not presently support them. To accurately compare the performance of the different binaries, simply timing a fixed number of instructions is incorrect. Instead, simulation markers are inserted in the code of each binary, and simulations are run for a given number of markers. After skipping the initialization (typically 1-6 billion instructions), a certain number of markers are executed, so that the baseline binary graduates from 500 million to 1 billion instructions.

6.1 Performance

Figure 7 shows the speedup on our evaluated applications over Base. The geometric mean of speedup for TLS+PRV is 1.31 for the CINT applications and 1.57 for the CFP. Compared to TLS, CINT is better on our proposed system, on average, by 5.84% and, CFP by better than 15.82%. These speedups were attained mostly using a fully-automatic compiler approach and are significant.

Figure 8 shows the fraction of wasted work for TLS and TLS+PRV. This is calculated as the number of squashed instructions (due to dependence violations) out of the total number of committed instructions. Since TLS+PRV removed some cross-thread dependences by handling PRVs, it should have fewer dependence violations and, thus, have a lower waste rate than TLS. For most applications and for the average behavior for both CINT and CFP, we see such results.

Table 5 provides some characterization of the reduction variables that are included in the finally survived speculative tasks. For each application, the reduction variables are classified as RV or PRV. Note that most reduction variables detected are PRV – except 2 significant RVs in parser (the 4 RVs in mcf don’t speedup performance at all). That is why we did not show the speedup of TLS+RV in Fig. 7 – because it is almost the same as TLS, except that parser has the same speedup with that of TLS+PRV. The last column of the table 5, together with Fig. 7 and Fig. 8, explains how these reductions affect performance. Most of the gains of our approach come from PRV tasks selected in twolf, vpr, art, and mesa. Not surprisingly, these applications contain tasks that had no loop-carried dependences other than the PRVs. Once the PRVs were optimized, the tasks were parallelizable and provided significant performance gains. The loop from twolf described in Section 2 is such a case. Impressively, identifying two PRVs was enough to provide the large performance gains in twolf. Note that the waste rates for vpr and mesa are considerably lower with PRV tasks. That’s because the PRVs introduce major dependences in these tasks. Once these dependences are handled, the task selector prefers these tasks which are quite different from those of TLS. For twolf, more waste occurs on the TLS+PRV tasks than on the TLS tasks. Without PRV support, TLS selects a set of tasks which speculate less aggressively. For TLS+PRV, the additional speculation opportunities brought by PRVs brings the benefit of higher performance at the cost of higher waste.

In parser, gzip, and mcf, although PRV/RV are detected and handled, they don’t boost performance much because either they are located in secondary or small tasks (gzip) or there still exist nontrivial cross-thread dependences other than PRVs (mcf). However, we can still notice the reduction in wasted work in Fig. 8 after handling the PRVs.

In crafty, gap, ammp, and equake, although many PRV/RV are detected (see table 1), none of them survives profiling, thus there is nearly no performance gain. However, performance of crafty and ammp is slightly affected due to the selection of a different set of tasks. This is because the early phase of compilation handles all PRVs – this affects the behaviour of tasks in which PRVs are located and leads to a different profiling result.

The third column of Table 5 classifies the reductions from another aspect. Local means the RV/PRV is a locally declared variable whose address is not taken (thus has no alias); Ptr means it is a pointer variable; Global means it is a global variable, and AddrTaken means it is a local variable with its address taken. Note that (1) the latter three cases may have aliases and need the support of compiler-hardware combined mechanism we proposed; (2) the Local PRVs are not traditional reductions and need our compiler schemes to handle them; (3) only parser contains significant RVs that can be handled by traditional techniques, but its performance gain is limited. Such a distribution of reductions shows the importance of handling PRVs (not just RVs) when parallelizing sequential codes.

6.2 Discussion

While many PRVs were identified by the compiler, there were many fewer tasks containing PRVs that were ultimately profitable. By examining the code that excluded these PRVs, we identified some key reasons. (1) Many PRV tasks in gap and gzip have small loop sizes and thus do not overcome the initial cost of speculation. For this reason, the compiler eliminates them during profiling. We believe that loop unrolling, if applied judiciously, can help generate some good PRV tasks by increasing the task size. (2) Some PRVs appear only on a branch which is seldom taken. Supporting these PRVs may add synchronization when not needed. Finding ways to reduce this cost and generate better PRV code help some cases found in gzip. (3) Some PRVs occur in tasks with other frequently occurring dependences, preventing the task from being selected. Incorporating additional techniques that target non-PRV dependences could increase the value of our mechanism.

Furthermore, some legitimate PRVs are missed by our compiler pass. Note, our algorithm requires that the update occur on a single variable. Consequently, we miss the classic array-reduction $A[i] = \ldots$ if $i$ is the induction variable for the innermost loop in the surrounding loop nest. Some preliminary analysis suggests
that many such PRVs exist in this form even in SPEC applications and many of them are located in potentially good tasks. We believe that extending our techniques to support such PRVs is possible and would increase the performance of our techniques.

7. Related Work

Reductions are a type of recurrence that are easy to parallelize on multiprocessors. They have been well researched in the context of optimizing compilers. The work in this area focused on a variety of issues (loosely categorized as follows) including detection of reductions [1, 4, 12, 17], parallelization/scheduling strategies [9, 18, 22, 27, 28, 34, 35], speculative approaches [6, 7, 21, 39], and architectural support [10]. The earlier techniques were often limited to loops in which the reduction variables and operators were fully analyzable by the compiler (even if the loop bounds were unknown). However, given the frequency of array based reductions and the limitations of dependence analysis on indirect array references, many loops could be optimized with this technique.

More recent work has identified the importance of employing speculation to extract more parallelism from hard-to-analyze reductions. Rauchwerger and Padua proposed the LRPD test [21] as a way of overcoming the limitations of static analysis. Instead of requiring a complete static analysis, some disambiguation tests were delayed until runtime. Their scheme works by first creating private versions of each array participating in the reduction computation, and calculating read and write sets for each array at runtime. Their technique is able to detect if the operations were in fact reductions during the actual execution, even though static analysis could not verify it either due to control or aliasing. Refinements of the LRPD algorithm aim to increase coverage and reduce overhead of this approach [7]. Many of the arrays classified by the LRPD scheme would be classified as PRVs by our scheme. However, these mechanisms require full control-flow analysis of the code being parallelized and identification of all loads and stores that may participate in the reduction. Non-analyzable control flow (e.g., into libraries) with unknown reads/updates cannot be treated with this mechanism. For nested loops which dominate execution time, this limitation is reasonable, however, for many C programs, these restrictions are prohibitive. Instead, we need an approach that can cope with potentially unknown reads or writes to the reduction variable at runtime. Our proposed scheme can tolerate such reads or writes to the reduction variable by monitoring all accesses to the variable and taking corrective action when such an update occurs. The proposed mechanism does not require the insertion of dependence tracking and tests to validate the reduction and can handle unknown paths and references never analyzed by the compiler.

Hardware support for reductions has been proposed in the past that extends beyond support for efficient synchronization primitives. Garzaran et al. proposed PCLR, hardware support for reductions that accelerates the merging phase of the reduction after the parallel region is complete. Their approach allows reductions to complete efficiently and lazily by combining the partial results of a reduction operation in hardware at the directory controller rather than in software. Our hardware support also performs merging, but that is not its primary responsibility. Our hardware support is focused on monitoring accesses to the reduction variable and taking corrective action when needed. Our approach is largely orthogonal to PCLR and could be integrated with aspects of PCLR to support efficient merging on highly parallel codes. Past work on support for efficient synchronization primitives is also orthogonal to our scheme and could be leveraged to reduce sync time.

Our definition of PRV shares many similarities with that described by Zhang et al. in UPAR[40]. However, UPAR only evaluates certain kinds of PRV. Our approach broadens the definition; it tolerates explicit accesses to the PRV, not just aliases. In addition, we search for PRVs in pointer-intensive integer codes, and our approach is better suited to find a wide variety of PRVs in this domain.

Compilers and systems for Thread-Level Speculation have identified the need to effectively handle reduction variables. Zhai et al. [39] show the benefit of reductions for SPECint applications and shows modest gains. However, this mechanism employed the strict definition of reduction variable, not the more flexible definition of PRV proposed in this paper. Also, Prabhu et al. [19] identifies reductions as an important transformation to unlock the potential of key loops in vpr, mcf, and twolf. Since they focused on manual techniques they transformed some reductions manually that would be classified as PRV in our approach. Finally, work in Thread-Level Speculation targeting efficient synchronization of cross-thread dependences [37, 38] is also relevant, since they are an alternative, but less direct, mechanism for supporting reduction variables.

8. Conclusion

In this work, we considered a kind of shared variable that may behave like a reduction at runtime, called partial reduction variables. PRVs differ from RVs by allowing reads and writes to the reduction variable outside of the reduction update operation. We found that PRVs are more frequently occurring than RVs in SPEC CPU 2000 applications. Given the frequency of these variables, it is important to consider hardware and software mechanisms that can exploit them. We implemented a parallelization framework for PRVs using an automatic parallelizing compiler for Thread-Level Speculation and a new architecture with necessary supports for detecting and correcting updates to PRVs that contradict its usual reduction like behavior. On a set of SPEC 2000 applications, we found that supporting PRVs provides up to 46% performance gain over one of the state-of-the-art TLS systems and on average a 10.7% improvement.

There are still many important optimization opportunities remaining for PRVs in highly irregular and integer codes. Depending on the reference or control structure that leads to detection of a PRV instead of RV, PRVs can be further classified into a variety of types. Parallelization strategies could be tailored for each kind of PRV to enable the most parallelism. Furthermore, PRVs can also be important in a variety of other systems. For example, dynamic optimization environments are often limited in the kinds of parallelization strategies that can be applied.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>RV</th>
<th>PRV</th>
<th>Local</th>
<th>Ptr</th>
<th>Global</th>
<th>Addr.Taken</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>crafty</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No PRV survives.</td>
</tr>
<tr>
<td>gzip</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No PRV survives. TLS and TLS+PRV select same tasks.</td>
</tr>
<tr>
<td>mcf</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>They are located in 2 tasks, but doesn’t speedup the performance due to other dependences.</td>
</tr>
<tr>
<td>parser</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>They are located in 2 tasks, from which the performance gain mostly comes</td>
</tr>
<tr>
<td>twolf</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td>TLS doesn’t select these 2 tasks due to dependences on these PRVs.</td>
</tr>
<tr>
<td>vpr</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>It is located in 1 task, from which the performance gain mostly comes</td>
</tr>
<tr>
<td>ammp</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No PRV survives.</td>
</tr>
<tr>
<td>alp</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>They are located in 3 tasks, from which the performance gain mostly comes</td>
</tr>
<tr>
<td>eqlake</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No PRV survives. TLS and TLS+PRV select same tasks.</td>
</tr>
<tr>
<td>mesa</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>Located in 1 task, from which the performance gain mostly comes. Task selection affected</td>
</tr>
</tbody>
</table>

Table 5. PRV Characterization.