Abstract

Harnessing the hardware parallelism of the emerging multi-cores systems necessitates concurrent software. Unfortunately, most of the existing mainstream software is sequential in nature. Although one could auto-parallelize a given program, the efficacy of this is largely limited to floating-point codes. One of the ways to alleviate the above limitation is to parallelize programs, which cannot be auto-parallelized, via explicit synchronization. In this regard, efficient placement of the synchronization primitives – say, post, wait – plays a key role in achieving high degree of thread-level parallelism (TLP). In this paper, we propose novel compiler techniques for the above. Specifically, given a control flow graph (CFG), the proposed techniques place a post as early as possible and place a wait as late as possible in the CFG, subject to dependences. We demonstrate the efficacy of our techniques, on a real machine, using real codes, specifically, from the industry-standard SPEC CPU benchmarks, the Linux kernel and other widely used open source codes. Our results show that the proposed techniques yield significantly higher levels of TLP than the state-of-the-art.

Categories and Subject Descriptors D.1 [Software]: Concurrent Programming—Parallel Programming

General Terms Algorithms, Performance

Keywords Multithreading, Parallelization, Compilers, Performance

1. Introduction

Multi-core systems are becoming ubiquitous. Exploitation of the hardware parallelism of such systems is critically dependent on the availability of concurrent software. One way to parallelize programs which cannot be auto-parallelized is via explicit thread synchronization, wherein dependences between the different concurrent threads are preserved with the help of synchronization primitives such as post and wait. Several approaches have been proposed for explicit synchronization [1, 12]. However, the existing approaches do not address the problem of efficient placement of the post and wait primitives. Consequently, synchronization primitives are typically placed at their “natural” positions – wait before the first read of a shared variable and post after the last write to the shared variable. This is exemplified by the loop (extracted from the Linux kernel [12]) shown in Figure 1, wherein spin_lock is placed before the first read of the field pmc->mca_sfcnt [MCAST_EXCLUDE] of an element of the list mc_list and spin_unlock is placed after the last write to the field mca_crcnt of an element of mc_list.1 However, placement of the synchronization primitives at their “natural” positions may and often does limit the exploitation of TLP. For instance, let us revisit the example shown in Figure 1, wherein each element of the list mc_list has its own lock. Although threads executing different iterations of the loop can potentially acquire their respective locks at the same time, the iterations of the loop cannot be executed in parallel. This can be ascribed to the recurrence based on the variable skb. The calls to spin_lock and spin_unlock in the original source code forbid concurrent access to the elements of the list mc_list by threads executing other code, thereby avoiding data races; however, the calls do not preserve the aforementioned recurrence which limits the execution of the iterations of the loop to a serial fashion.

To this end, first we introduce post, wait (see Figure 1) in the loop body and then optimize the placement of the synchronization primitives to facilitate efficient parallel execution of the loop (the lines 1735–1746 after optimization are shown on the right).

Essentially, the problem of efficient placement of the synchronization primitives – post and wait in the current context – has the following dual objectives: (a) how to place a post as early as possible in the CFG and (b) how to place a wait as late as possible in the CFG. Arguably, the primitives can be placed optimally manually. However, this is not pragmatic and is error-prone owing to the high complexity associated with carrying dependence analysis of modern applications which span over millions of lines of code. Clearly, there is a need for an automatic approach to guide the placement of the synchronization primitives. In this paper, we address the above. In particular, the main contributions of the paper are:

1. We propose compiler techniques for efficient placement of the post and wait synchronization primitives. Specifically, the

---

1 Note that spin_lock and spin_unlock are the counterparts of wait and post respectively.
proposed techniques percolate/move the post and wait primitives upwards and downwards respectively, thereby exposing higher level of TLP.

Several techniques have been proposed for code motion for program optimization [13, 14, 15, 16, 17]. The key differences between these and the proposed techniques are the following:

1. Code motion induced by the existing techniques is primarily geared towards either elimination of redundant computation (e.g., [18]) or exploitation of instruction-level parallelism (e.g., [13, 14, 15]). In contrast, the proposed techniques induce code motion in order to minimize the effect of ordering, induced by the post, wait primitives, between the different threads.

2. The proposed techniques support downward percolation of post primitive. Furthermore, we show that upward percolation does not necessarily affect the same code motion as downward percolation.

3. We present results to demonstrate the efficacy, w.r.t. performance, of the proposed techniques using kernels extracted from the industry standard SPEC CPU2000, CPU2006 benchmarks [19], the Linux kernel and other widely used open source codes such as sendmail. The optimized kernels were compiled using the Intel C++ compiler and executed on a real machine.

Given that loops account for a large percentage of the total execution time in real programs [20], we focus on parallelization of non-DOALL loops, i.e., loops in which there exists one or more loop-carried dependencies [21], via explicit synchronization.

The rest of the paper is organized as follows: The motivation behind the problem addressed in this paper is presented in 2. Section 3 introduces the terminology used in the rest of the paper. Section 4 walks through a real-life example, extracted from a SPEC CPU2006 [22] application, to illustrate the different kinds of code motion enabled by the techniques presented in Section 5 to achieve best placement of the synchronization primitives. The results are presented in Section 6. An overview of related work is presented in Section 7. Finally, Section 8 concludes with directions for future work.

2. Synchronization Placement: Why Bother?

In the context of non-DOALL loops, the problem of synchronization placement is to determine points in the loop body for placing the post and wait primitives so as to: (a) preserve the program semantics during multithreaded execution of the loop and (b) minimize the “coupling-effect” (discussed later in this section) between the different threads. The latter is necessary to extract high degree of TLP. From hereon, we represent the call to the synchronization primitives post and wait by the symbols \( \bullet \) and \( \triangle \) respectively and all the other operations by the symbol \( \triangledown \).

Let us consider the scenario shown in Figure 2. A solid line in the figure represents the loop body. For simplicity of exposition, we consider a two thread case in the rest of the section. Each thread is divided into segments delimited by either the post or the wait primitive. From the figure we see that segments \( S_1, S_3 \) can be executed in parallel with no restrictions. However, the execution of segment \( S_1 \) is dependent on the completion of segment of \( S_2 \). We refer to this dependence between segments of the different threads as “coupling”. Note that the execution of the first segment of any thread is not dependent on any other segment. If there is a post/wait primitive at the top of the loop body, then the first segment is empty. If a segment starts with the post primitive, it can be executed as soon as its preceding segment has finished execution. From Figure 2 we note that percolating the wait primitive on thread \( T_2 \) downwards would shorten the length of the dependent segment \( S_4 \), denoted by \( \ell(S_4) \), which in turn reduces the coupling between the two threads.

![Figure 2. Illustrating the impact of pipeline bubbles](image)

Next, we analyze the coupling between the threads \( T_1 \) and \( T_2 \) when \( \ell(S_1) \approx \ell(S_3) \ll \ell(S_2) \) (see Figure 3 (a)). Based on the short lengths of the segments \( S_1 \) and \( S_3 \) one may expect an early execution of the post, wait primitives. This may give an impression of a weak coupling between the threads \( T_1 \) and \( T_2 \). However, this is not necessarily true as exemplified by Figure 3 (b). From the figure we see that a pipeline bubble prior to the execution of the post primitive will result in idling of thread \( T_2 \). In order to mitigate the effect of this, the wait primitive should always be placed (in the loop body) as late as possible.

![Figure 3. Illustrating the impact of pipeline bubbles](image)

2.1 Placement Scenarios

In the vanilla case, the post, wait primitives can be placed by simply identifying the operations corresponding to the loop-carried dependence(s). However, this may not result in the best placement of post, wait. The latter stems from the fact that the operations may be percolated upwards/downwards which may in turn result in better placement, compared to the vanilla case, of the post and wait primitives.

Figure 4 illustrates the different scenarios corresponding to the placement of the post, wait primitives. For simplicity of exposition we only consider a pair of post, waits. We classify the placement of the post, wait primitives into the following two categories:

- **Case I**: A post is placed above a wait. In such a scenario, the upward percolation of post and the downward percolation of wait are decoupled from each other. In other words, the former does not limit the latter and vice-versa.

- **Case II**: A wait is placed above a post. In such a scenario there are two possibilities:
  a) There does not exist a true dependence between the wait and the post.\(^2\) Thus, the post can be percolated upwards beyond the wait (see Figure 4 Case II (a)).
  b) There does exist a true dependence between the wait and the post. Code motion in such a scenario involves an interesting trade-off.

\(^2\) An anti-dependence can be eliminated via renaming, as illustrated in Figure 6.
(i) Scheduling the post as early as possible results in an early scheduling of the wait. This may in turn induce “false” partial ordering between the operations of different iterations.

(ii) Scheduling the wait as late as possible delays the scheduling of the post. This would in turn delay the scheduling of the operations after the call to the post primitive.

The trade-off arises only when the wait-post segment does not lie on a critical path – a critical path is the longest path in a DAG [23] – of the data dependence graph of the given loop, ignoring the loop-carried dependences. In the context of the running example, the conflict arises due to the flow dependence, based on B[i], between wait and post. However, one can potentially compact the wait-post segment subject to dependences.

### 3. Terminology

Let $N$ denote the set of nodes $n_0, n_1, \ldots$ in a flow graph; node $n_0$ is the start node. We use two set-valued functions PRED and SUCC, such that for each $n \in N$, PRED($n$) is the set of all immediate predecessors of $n$ and SUCC($n$) is the set of all immediate successors of $n$. At any node $n_j$, we denote the incoming edges by $I_j$ and exiting edges by $E_j$.

Execution begins at the start node and proceeds sequentially from node to node. When the control reaches a particular node, all operations in the node are evaluated concurrently; the assignments update the registers or memory locations and the conditionals return the next node in the execution sequence. Operations evaluated in parallel perform all reads before any assignment performs a write. Write conflicts within a node are not permitted.

A node may contain at most one conditional initially; however, as operations are percolated up (explained further in Section 5), a node may contain multiple conditionals. We model the set of conditionals in a node as a directed acyclic graph (DAG) [24]. Each conditional in the DAG has two successors corresponding to its true and false branches. Further, a successor of a conditional is either another conditional or a pointer to a node. The DAG of conditionals is assumed to be rooted, i.e., it has a single element with no predecessors. To evaluate a DAG in a node, a (unique) path from the root to a leaf node is selected such that the branches on the path correspond to the value (true or false) of the corresponding conditionals on the path. Evaluation of the DAG returns the node that terminates this path.

Given a DAG of conditionals, we define three set-valued functions: $s_t(x)$ denotes the set of operations above a conditional $x$, $s_f(x)$ denotes the set of operations on the true branch of $x$, and $s_b(x)$ denotes the set of operations on the false branch of $x$. For example, consider the node shown in Figure 5. Node $n$ consists of a DAG of conditionals, where $x$ is a conditional operation and $a, b, c$ are DAG of conditionals themselves. In this case, $s_p(x) = a$, $s_t(x) = b$ and $s_f(x) = c$.

### 4. Synchronization Placement in Real-Life

Let us consider the loop shown in Figure 6 (a). The loop is taken from 464.c264ref:block.c:632, a benchmark in the SPEC CINT2006 suite [26]. On analysis we observe that the loop is a non-DOALL loop owing to the following:

- There is a recurrence based on the variable run, shown by dashed arrows in the figure. Note that the recurrence cannot be parallelized via reduction.
- There is a recurrence based on the variable scan_pos.
- There is a potential output dependence between the write to the array DCLevel in the different iterations.

There is no aliasing between the writes to the array M4 in the different iterations. This is due to the fact that for each value of the iterator coeff_ctr, the pair $(i, j)$ has a different set of values. The set of values for $(i, j)$ are known at compile time. For example, FIELD_SCAN is defined as follows (block.h:41):

```c
const byte FIELD_SCAN[16] = {
  {0,0}, {0,1}, {1,0}, {0,2},
  {0,3}, {1,1}, {1,2}, {1,3},
  {2,0}, {2,1}, {2,2}, {2,3},
  {3,0}, {3,1}, {3,2}, {3,3}
};
```

Let $v^k_i$ denote the $k$-th operation in the $i$-th iteration and let iterations $i$ and $i+1$ be mapped on to threads $T_1$ and $T_2$ respect-
Parallel execution of the loop in the absence of explicit synchronization may violate the program semantics as \( v_{20} \) may get executed before \( v_{19} \). To orchestrate the execution of threads in the presence of dependences, synchronization primitives – post, wait – are inserted in the loop body. The “natural” placement of post, wait is marked in the shaded boxes in Figure 6 (a). The wait in iteration \( i \) + 1 suspends the execution of \( T_2 \) until the post in iteration \( i \) has been executed by \( T_1 \).

From the figure we note that the wait is placed “early” in the loop body which limits parallel execution of operations \( v_8, \ldots, v_{20} \) in (say) iterations \( i \) and \( i + 1 \). This is due to the ordering induced by post in iteration \( i \) and wait in iteration \( i + 1 \). Likewise, the post is placed “late” in the loop body which also limits the exploitation of TLP.\(^3\) In light of this, we pose the following questions in general:

a) Can we percolate the wait downwards?

b) Can we percolate the post upwards?

To this end, we present novel techniques which alleviate the above limitations. In the context of the running example, the code motion induced by our techniques is shown in Figure 6 (b). In particular:

- Wait has been percolated below operation \( v_{15} \). This enables parallel execution of operations \( v_9, \ldots, v_{15} \) of different iterations.

- Block \( B_2 \) has been percolated upwards beyond \( v_8 \). The validity of the code motion stems from the fact that block \( B_2 \) is independent of block \( B_1 \) (see Figure 6 (a)), subject to memory renaming. The “false” dependence between \( v_{17} \) and \( v_{23} \) is eliminated by copying \( M4[i][j] \) into the variable \( \text{tmp} \) (see Figure 6 (b)) and passing \( \text{tmp} \) to \( v_{17} \). Block \( B_2 \) is then percolated up beyond operation \( v_8 \). The proposed techniques support percolation of both conditionals and non-conditional. For better efficiency of the synchronization placement process itself, "trailblazing" [27] is employed to support hierarchical percolation of program regions.

Arguably, block \( B_2 \) can be percolated up even further, say, along the else branch of the conditional \( v_{13} \). However, this is unnecessary from TLP extraction perspective because in the transformed loop, block \( B_1 \) in the different iterations can be executed in parallel. Admittedly, further upward percolation of block \( B_1 \) may yield higher levels of instruction-level parallelism (ILP); however, this is out of scope of this paper.

Observe that the number of operations below \( \text{wait} \) is reduced from 16 to 6 using our techniques. In the next section we detail our techniques for synchronization placement.

5. The Techniques

The problem of placement of the synchronization primitives gives rise to the following questions:

- Which operations to percolate and in which direction – upwards or downwards?
- How to percolate an operation? Specifically, what are the rules that should be followed (during code motion) to guarantee program correctness?
- How “far” (up or down) should an operation be percolated?
Given a non-DOALL loop, the set of post(s) and wait(s) inserted at their “natural” positions constitute the set of operations to be percolated. There may exist multiple posts and waits in the CFG corresponding to the different paths in the CFG. The posts are percolated upwards whereas the waits are percolated downwards. If dependences prevent either of the above, then corresponding operations are “recursively” percolated upwards and downwards respectively. Assists such as memory renaming are invoked on a demand-driven basis to eliminate spurious dependences (as in the example discussed in Section 4) which inhibit code motion.

We present a set of transformations to drive the percolation of the post, wait synchronization primitives. The transformations operate on adjacent nodes in the CFG and are applied iteratively. The reason behind the latter is that the application of one transformation may expose opportunities for further code motion using another transformation (illustrated later in Figure 12). Although the iterative application of these transformations allows the operations to percolate to the top of the CFG, subject to dependences, in cases corresponding to Case II(b) described earlier in subsection 2.1, we limit the upward code motion up of operations other than the synchronization primitives to the wait placed highest in the CFG; likewise, we limit the downward code motion of operations other than the synchronization primitives up to the post placed lowest in the CFG. For example, in Figure 6, although block B2 could be percolated above operation v13, its upward percolation is limited to beyond the wait primitive. This is done to contain code explosion. Also, further upward/downward code motion does not expose high level of ILP (this is explained further later in this section); albeit, it can potentially expose higher level of ILP, however, this is orthogonal to the current context. Note that the highest (lowest) position of the wait (post) may change during iterative application of the proposed transformations. The only restriction placed on the transformations is that of respecting data dependences, which preserves the sequential execution semantics of the original program. Next, we detail the set of transformations to gear placement of the synchronization primitives. The transformations presented herein build upon the core transformations of percolation scheduling (PS) [28]. Conceivably, the proposed techniques could also be grafted on top of any other ILP techniques [13, 29]. We chose PS owing to its provable generality of code motion [14].

5.1 The “Simple” Case

In this subsection we present a technique for guiding the placement of synchronization primitives in loops such as in Figure 1, i.e., loops with no conditionals in the loop body. Specifically, we present a transformation – referred to as Move-Op – for percolating an assignment operation upwards or downwards. Based on the direction of code motion, we sub-classify the transformation into Move-Op-Up and Move-Op-Down. We describe the former in detail and present an illustrative real-life example. Move-Op-Down induces code motion in a similar fashion to Move-Op-Up, albeit in the downward direction. However, code motion induced by one does not subsume the other, as shown at the end of this subsection.

Move-Op-Up moves an assignment operation $x$ from a node $n$ to a node $m$ along the edge $(m, n)$ subject to the following: (a) no conflict exist between the operations in $n$ and the operations in $m$, (b) $x$ does not kill any live value [30] at $m$ and (c) $x$ does not write to a shared memory location which is written along any path passing through $m$ but not through $n$ (this is the key differentiating aspect between Move-Op-Up and Move-Op proposed in [14]; further, the importance of this constraint with respect to avoidance of data race is explained later in subsubsection 5.1.2). Care should be taken not to affect the computation along the paths passing through $n$ but not through $m$. To ensure this, the original node $n$ is copied along all such paths. The transformation is shown in Figure 7 wherein the assignment operation $x$ is moved from node $n$ to node $m$; furthermore, in order to preserve the semantic correctness, node $n$ is duplicated ($n''$) along the path corresponding to the incoming edge $E_2$.

Arguably, the duplication of $n$ to $n''$ in Figure 7 can potentially be prohibitively expensive if $n$ comprises of a large number of operations. One way to alleviate this is to insert an empty node $n'''$ such that $I_2$ points to $n''$ and $\text{SUCC}(n''') = n$ and copy $x$ in $n'''$.

![Figure 7. Move-Op-Up Transformation](image)

## Figure 7. Move-Op-Up Transformation

The Move-Op-Down transformation is an “inverse” of Move-Op-Up. Move-Op-Down is shown in Figure 8 wherein the assignment operation $x$ is moved from node $n$ to node $m$. In order to preserve the semantic correctness, node $n$ is duplicated ($n'$) along the path corresponding to the outgoing edge $E_1$. The downward percolation of $x$ is subject to the following: no conflict exists between the operations in $n$ and the operations in $m$, $x$ does not kill any live value at $m$ and $x$ does not write to a shared memory location which is written along any path passing through $m$ but not through $n$.

5.1.1 Move-Op-Up ⇔ Move-Op-Down

Let us consider the data dependence graph shown in Figure 9. From the figure we see that the wait can be scheduled in parallel with ei-
ther of the following: \( \{v_2, v_3, v_4, v_5\} \). The schedules obtained using \textit{Move-Op-Up} and \textit{Move-Op-Down} are shown in Figures 9(b) and (c) respectively. In Figure 9(b) the \texttt{wait} is scheduled in parallel with \( v_2 \) (the upward percolation of the \texttt{wait} is shown with a dotted arrow). This induces a “false” partial ordering between the execution of the operations \( v_3, v_4, v_5 \) between two iterations. In other words, \textit{Move-Op-Up} does not guarantee the placement of a \texttt{wait} primitive as late as possible. On the contrary, the \texttt{wait} is scheduled in parallel with \( v_5 \) in Figure 9(c) (the downward percolation of the \texttt{wait} is shown with a dotted arrow). This corresponds to the latest position the \texttt{wait} can be scheduled.

The above highlights that both \textit{Move-Op-Up} and \textit{Move-Op-Down} are required to guarantee best placement of the \texttt{post}, \texttt{wait} primitives! If the \texttt{wait-post} segment lies on a critical path of the data dependence graph of the given loop, then both \textit{Move-Op-Up} and \textit{Move-Op-Down} would effect the same placement of \texttt{post}, \texttt{wait} primitives!

5.1.2 Discussion

Conceivably, \textit{Move-Op-Up} can be extended to mimic \textit{Move-Op-Down}, as shown in Figure 10.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure10.png}
\caption{Extended \textit{Move-Op-Up}}
\end{figure}

Let us revisit Figure 9(a). From the figure we see that the upward percolation of \( \{v_3, v_4, v_5\} \) to the node containing the \texttt{wait} is limited by the dependence on \( v_2 \). Also, \( v_2 \) cannot be percolated upwards due to its dependence on \( v_1 \). To enable the upward percolation of \( v_3 \), we introduce an empty node between the nodes containing \( v_1 \) and \( v_2 \) as shown in Figure 10(b). Then, \( v_2 \) is percolated to the empty node and \( v_3 \) is percolated to the node containing the \texttt{wait}, see Figure 10(c). This procedure is repeated until \( v_4 \) is percolated above the \texttt{wait}. The final schedule is shown in Figure 10(d) which is the same as Figure 9(c). Thus, extended \textit{Move-Op-Up} can be used to drive downward percolation. However, \textit{Move-Op-Up} is non-intuitive as it is not obvious when to introduce an empty node. Therefore, we use \textit{Move-Op-Down} to drive downward percolation of a \texttt{wait} primitive.

Lastly, Figure 11 highlights the difference between code motion in the context of sequential execution and multithreaded execution. From Figures 11(a) and (b) we observe that the \texttt{post} along the false branch of the conditional can be percolated above the operation \( v_2 \), assuming that \( A[i] \) is not read before the \texttt{post} along the branch. The facilitates an early write to \( A[i] \) along the false branch. The above is valid for sequential execution, assuming that \( A[i] \) is not read before the \texttt{post} along the true branch, but can potentially result in data races during multithreaded execution. This is illustrated in Figure 11(c), wherein we assume that thread \( T_2 \) takes the false branch of the conditional \( v_3 \). An early execution of the \texttt{post} by thread \( T_2 \) may result in \( A[i]=\texttt{true} \) (instead of \( A[i]=\texttt{false} \) being read by operation \( v_3 \) on thread \( T_2 \)) recall that the threads execute asynchronously with respect to each other), which corresponds to a data race. Therefore, in the context of upward percolation of a \texttt{post} we forbid such type of code motion. Note that if the write to \( A[i] \) is not read in a subsequent iteration (in general, the write is private to the current iteration), then the code motion illustrated in Figure 11(b) is permitted, as in the case of vanilla \textit{Move-Op} transformation of percolation scheduling [28].

In contrast, the code motion described above is permitted for delaying a \texttt{wait} along a given path, subject to the preservation of the program semantics during sequential execution. This does not result in data races but would cause execution of a redundant \texttt{wait} along the other path(s).

To summarize: (a) \textit{Move-Op-Up} is used for upward percolation of a \texttt{post} and the operations it is dependent on (b) \textit{Move-Op-Down} is used for downward percolation of a \texttt{wait} and the operations that depend on it.

5.2 Handling Conditionals

Recall that \textit{Move-Op-\{Up/Down\}} transformations do not support percolation of conditionals. This can limit the upward and downward percolation of the \texttt{post} and \texttt{wait} respectively, as shown in Figure 12.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure12.png}
\caption{Illustration of upward percolation of conditionals}
\end{figure}

From Figure 12(a) we observe that \texttt{post} cannot be percolated to the node containing the conditional \( v_2 \) as it would kill the value of \( A[i] \) being read by \( v_3 \). This delays the execution of the \texttt{post} after the execution of \( v_1 \). However, if \( v_2 \) is percolated above \( v_1 \) then the \texttt{post} can be executed in parallel with \( v_1 \) (see Figure 12(b)). In order to address such scenarios, we present a transformation, referred to as \textit{Move-Test}, for upward/downward percolation of conditionals.

The \textit{Move-Test} transformation is shown in Figure 13, where \( a \) represents a DAG of conditionals not reached by \( x \). \( b \) represents a DAG of conditionals reached on \( x \)’s true branch, and \( c \) represents a DAG of conditionals reached on \( x \)’s false branch. \textit{Move-Test} percolates the conditional \( x \) upwards from node \( m \) to node \( n \) along the edge \( \langle m, n \rangle \) provided that no dependency exists between \( x \) and the operations of \( m \). The conditional being moved up may come from an arbitrary point in a DAG of conditionals. In order to preserve the program semantics, the transformation does the following:
It copies \( n \) on the paths passing through \( n \) but not through \( m \). In Figure 13, the node \( n \) is copied \((n')\) on the path corresponding to the incoming edge \( I_2 \).

It “splits” the node \( n \) into two nodes \( n_t \) and \( n_f \), where \( n_t \) and \( n_f \) correspond to the true and false branches of \( x \), and copies the DAG of conditionals \( a \) in \( n_t \) and \( n_f \).

The Move-Test transformation may catalyze further (i.e., beyond what can be achieved via Move-Op in isolation) upward/downward percolation of \( \text{post} \), \( \text{wait} \), \( \text{post} \), \( \text{wait} \) conditionals and other operations and assist in further compaction of the \( \text{wait-post} \) segment (described in Section 2.1). Next, we present the algorithm for the Move-Test transformation.

### Transformation Move-Test \((v, m, n)\).

Let the false branch edge of a conditional \( x \) be denoted by \( x_f \) and the true branch edge be denoted by \( x_t \).

```c
/* Duplicate n */
for each node \( p \in \text{PRED}(n) - \{m\} \) do
    Create a copy \( n' \) of node \( n \)
    \( \text{PRED}(n') \leftarrow p \)
    \( \text{SUCC}(n') \leftarrow \text{SUCC}(n) \)
endfor
/* Create nodes corresponding to the true and false branches */
\( n_t \leftarrow s_p(x) \cup s_t(x) \)
\( n_f \leftarrow s_p(x) \cup s_f(x) \)
Move \( x \) to the bottom of the conditional tree of node \( m \)
Delete \( n \)
Along \( x_t \), \( \text{SUCC}(x) \leftarrow n_t \)
Along \( x_f \), \( \text{SUCC}(x) \leftarrow n_f \)
```

Arguably, a conditional or a DAG of conditionals can be moved to the top of the program’s control flow graph (CFG), subject to unrestricted code duplication. However, in the context of extraction of TLP, we restrict the upward percolation of a conditional up to a sink of a loop-carried dependence placed highest in the CFG. This is done to limit code explosion induced by upward percolation of conditionals. For example, in Figure 13, the upward percolation of the conditional \( x \) is limited to node \( m \), if the operations in the nodes preceding \( m \) in the CFG do not induce a loop-carried dependence.

Next, we illustrate the application of the Move-Test transformation with the help of an example code taken from a real-life code.

**Example 1.** Let us consider the loop shown in the Figure 14(a), taken from 254, gap:costab.c:561. The macros in the loop body are defined as follows:

- \#define INT_TO_HD(INT) \((\text{TypHandle}) (((\text{long})(\text{INT}) \ll 2) + \text{T_INT})\)
- \#define HD_TO_INT(HD) \(((\text{long})(\text{HD}) \gg 2)\)
- \#define T_INT 1

A schedule before transformation is shown in Figure 14(b). The operations \( v_1 \), \( v_2 \) can be executed in parallel as there does not exist any data dependence between them. In contrast, the operations \( v_5 \), \( v_6 \) cannot be executed in parallel as there may exist an output dependence between \( v_5 \), \( v_6 \). The \( \text{wait} \) primitive is placed before the operations \( v_3 \), \( v_4 \), see Figure 14(b), to preserve the potential flow dependence between \( v_8 \) in iteration \( \ell \) and \( v_3 \) in iteration \( m \), where \( \ell < m \). Similarly, the \( \text{post} \) primitive is placed after \( v_7 \) to preserve the potential output dependence between \( v_7 \) and \( v_5 \) in iteration \( m \), where \( \ell < m \). Most importantly, before transformation, we observe that the scheduling of operations \( v_{11}, \ldots, v_{17} \) is serialized w.r.t. the scheduling of operations \( v_5, \ldots, v_9 \). Clearly, this delays the scheduling of the \( \text{post} \) primitive which adversely affects performance.

The schedule after applying the Move-Test transformation is shown in Figure 14(c). From the schedule we see that the conditional \( v_{10} \) has been percolated above the operations \( v_3 \) and \( v_4 \). This necessitates the duplication of the operations \( v_3, \ldots, v_9 \) along the false branch of the conditional. The upward percolation of the conditional \( v_{10} \) enables the compaction of the basic block along the false branch of \( v_{10} \). This implicitly effects upward percolation of the \( \text{post} \) primitive. Note that the operations \( v_{11} \) and \( v_{12} \) cannot be moved to node containing \( v_6 \) due to a potential aliasing of \( \text{i}[\text{c2}] \) with either (or both) of \( \text{i}[\text{lcos}] \), \( \text{i}[\text{mcos}] \). On comparing the two schedules, we note that the transformation reduced the schedule length from 16 steps to 11 steps—a reduction of 31%!

Another important aspect of the Move-Test transformation is that it facilitates “customized” compaction of the different paths. In the context of the running example, this is evidenced by the different position of, say, the operation \( v_{15} \) along the true and the false branches of \( v_{10} \) (see Figure 14(c)). Although \( v_{10} \) could be percolated to the top of the CFG, it would require duplication of operations \( v_1 \) and \( v_2 \). More importantly, the above is unnecessary as further upward percolation of \( v_{10} \) (than shown in Figure 14(c)) does not affect further compaction.

---

5 The dependence will materialize at run-time if the values of \( c_1 \) and \( c_2 \) are equal.
The code explosion incurred is very minimal due to the presence of small number of conditionals in loops (with high coverage, where coverage is defined as the percentage of the total run time) in real programs. A detailed analysis of the trade-off between gain in TLP vs. code duplication is beyond the scope of this paper. Note that the transformation itself is decoupled from any heuristic used for assessing the trade-off and thus, transformations which induce less code explosion may be used (at the expense of less general code motions).

5.3 Eliminating Copies

Iterative application of the Move-Op-{Up/Down} and Move-Test transformations may result in “redundant” copies of operations along the different paths of the CFG. To eliminate the copies, we propose the Unify-{Up/Down} transformation. The Unify-Up transformation moves a copy of identical assignments operations \( x \) from a set of nodes \( n_j \) to a common predecessor node \( m \). This is done if no dependency exists between \( x \) and the operations of \( m \), \( x \) does not kill any value live at \( m \) and \( x \) does not write to a shared memory location which is written along any path passing through \( m \) but not through \( n \). A node \( n_j \) is copied along each path passing through \( n_j \) but not through \( m \) so as to preserve semantic correctness. The Unify-Up transformation is illustrated in Figure 15.

The Unify-Down transformation is an “inverse” of Unify-Up, wherein a copy of identical assignments operations \( x \) from a set of nodes \( n_j \) to a common successor node \( m \). This is done if no dependency exists between \( x \) and the operations of \( m \), \( x \) does not kill any value live at \( m \) and \( x \) does not write to a shared memory location which is written along any path passing through \( m \) but not through \( n \). A node \( n_j \) is copied along each path passing through \( n_j \) but not through \( m \) so as to preserve semantic correctness. The transformation is shown in Figure 16.

5.4 Eliminating Redundant Operations

The Delete transformation proposed in [28] is used to remove a node from the CFG if it is empty (contains no operations) or is unreachable. A node may become empty or unreachable as a result of other transformations or elimination of redundant synchronization.
primitives. The transformation is illustrated in Figure 17, wherein the empty node \( n \) (represented by a dashed circle) is deleted from the flow graph. Note that an empty node has exactly one successor.

![Delete Transformation](Image)

Figure 17. Delete Transformation

6. Results

In this section, we demonstrate the efficacy of the techniques proposed in Section 5 using real codes. For this, we extracted kernels from the industry-standard SPEC CPU2000, CPU2006 [31, 22] benchmark suites, the Linux Kernel [12] (v2.6.23.1) and other open source applications such as sendmail (v8.14.3) [32] and Apache (v1.3.41) [33]. The details of the kernel set is given in Table 1. We picked kernels from a wide variety of benchmark sets on purpose, rather than concentrating on one benchmark set (e.g., SPEC CPU2006), in order to illustrate the wide applicability of the proposed techniques. Note that while in some of the kernels we inserted the synchronization primitives ourselves (as the original source code of the benchmark is not multithreaded), in others – e.g., IPVS – we used the synchronization placement in the original source code as our baseline. Evaluation of the proposed techniques on overall benchmarks is beyond the scope of this paper. The primary focus herein is to showcase a previously unexplored optimization opportunity, and provide evidence of its practical applicability in real codes using real hardware.

Table 1. Kernel Set

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Suite</th>
<th>Benchmark</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(gap)</td>
<td>SPEC CINT2000</td>
<td>254.gap</td>
<td>costabc</td>
</tr>
<tr>
<td>L(lucas)</td>
<td>SPEC CFP2000</td>
<td>189.lucas</td>
<td>lucas_intel SPEC190</td>
</tr>
<tr>
<td>L(vpr)</td>
<td>SPEC CINT2000</td>
<td>175.vpr</td>
<td>place.c</td>
</tr>
<tr>
<td>L2(vpr)</td>
<td>SPEC CINT2000</td>
<td>175.vpr</td>
<td>place.c</td>
</tr>
<tr>
<td>L3(vpr)</td>
<td>SPEC CINT2000</td>
<td>175.vpr</td>
<td>place.c</td>
</tr>
<tr>
<td>L1(bzip2)</td>
<td>SPEC CINT2000</td>
<td>401.bzip2</td>
<td>blocksort.c</td>
</tr>
<tr>
<td>L2(bzip2)</td>
<td>SPEC CINT2000</td>
<td>401.bzip2</td>
<td>compress.c</td>
</tr>
<tr>
<td>L1(l264ef)</td>
<td>SPEC CINT2000</td>
<td>464.l264ef</td>
<td>block.c</td>
</tr>
<tr>
<td>L2(l264ef)</td>
<td>SPEC CINT2000</td>
<td>464.l264ef</td>
<td>block.c</td>
</tr>
<tr>
<td>L(sjeng)</td>
<td>SPEC CINT2000</td>
<td>458.sjeng</td>
<td>see.c</td>
</tr>
<tr>
<td>L(ipv6)</td>
<td>Linux Kernel</td>
<td>ipv6</td>
<td>ip,ipv6est.c</td>
</tr>
<tr>
<td>L(sendmail)</td>
<td>Sendmail</td>
<td>sendmail</td>
<td>engine.c</td>
</tr>
<tr>
<td>L(apache)</td>
<td>Apache</td>
<td>Apache</td>
<td>os-axi-dso.c</td>
</tr>
</tbody>
</table>

Table 2. Experimental Setup

<table>
<thead>
<tr>
<th>System</th>
<th>Dual Core Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Architectural Dual CPU E2140 @ 1.60GHz</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>32 KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1024 KB</td>
</tr>
<tr>
<td>Memory</td>
<td>1025020 KB</td>
</tr>
<tr>
<td>Compiler</td>
<td>Intel C++ Compiler v10.1</td>
</tr>
<tr>
<td>Compiler Flags</td>
<td>-parallel -openmp -O3 -xN</td>
</tr>
<tr>
<td>Thread Library</td>
<td>NPTL 2.6.1</td>
</tr>
<tr>
<td>OS</td>
<td>Linux ubuntu 2.6.22-14-generic #1 SMP</td>
</tr>
</tbody>
</table>

Figure 18. Performance gain achieved via techniques proposed in Section 5

From the figure we see that optimization of the placement of the post, wait primitives via the techniques proposed in the earlier section yields performance gains up to 60.34%. Again, the gains reported above are w.r.t. multithreaded execution with vanilla synchronization placement; of course, the gains would be much higher with respect to sequential execution.

7. Previous Work

There is a large body of work in the areas of synchronization (at both the compiler and the operating system level), elimination of redundant synchronization, TLP exploitation, deadlock avoidance and sequential consistency memory model. To limit the scope of our discussion, due to space limitations, we focus our attention on the work done in context of explicit synchronization. The main focus of the existing techniques for the latter has been to reduce the synchronization overhead, such as [34, 35, 36, 37, 38, 39, 40]. Other work in the context of thread synchronization has primarily focused on the development of techniques for better pointer and escape analysis to minimize the need for thread synchronization [41, 42].

In [43], Cytron proposed to introduce delays in iterations of a given non-DQALL loop to preserve lexically backward dependences during parallel execution. The above assumed that the processors execute at approximately the same rate and also assumed availability of infinite number of processors. The former does not hold during multithreaded execution. Also, the above does not handle lexically forward dependences. Midkiff and Padua proposed several techniques for automatic generation of synchronization for D0
loops [44]. Later on, Kasahara et al. and Girkar proposed techniques for deriving conditions for explicit task synchronization in [45] and [46] respectively. None of the aforementioned techniques do not support for upward/downward code motion of the synchronization primitives which, as demonstrated in the previous section, plays a vital role in extracting higher degree of TLP.

In [47], Cytoron et al. proposed a technique for exploiting nested, fork-join parallelism. A join acts as a barrier to all the previously forked threads, thereby limiting exploitation of TLP. Subsequently, Sarkar proposed a technique for instruction-reordering for exploiting higher degree of parallelism in the fork-join execution model [48]. The techniques proposed in the above works are not applicable under the post, wait-based point-to-point synchronization model. This stems from the fact that in the post, wait model threads need not wait for each other to finish execution.

Recently, Tian et al. proposed a technique for dynamic recognition of synchronization operations in multithreaded programs [49] and showed its applicability for data race detection. They do not address the problem of efficient placement of synchronization primitives. Hence, their technique is complementary to the one presented in this paper.

8. Conclusion

In this paper we presented techniques to drive advance parallelization of non-DDALL loops that could not be parallelized very well using the existing techniques. We demonstrated the efficacy of our techniques using real codes, such as the Linux kernel, wherein the post, wait synchronization primitives are placed at their "natural" positions – which may and often is suboptimal – by experts (indeed those writing critical code and are very much interested in very efficient execution of their code), as well as other serial codes. The placement of the synchronization primitives was optimized using the proposed techniques. We achieved speedups up to 60.34% on a real machine.

As future work, we intend to develop techniques to capture data affinity while mapping program regions to the different cores of a multi-core system.

References