Extending Global Optimizations in the OpenUH Compiler for OpenMP

Lei Huang Deepak Eachempati Marcus W. Hervey Barbara Chapman

Computer Science Department
University of Houston
Houston, Texas 77004, USA
Email: leihuang, dreachem, mwhervey, chapman@cs.uh.edu

Abstract

This paper presents our design and implementation of a framework for analyzing and optimizing OpenMP programs within the OpenUH compiler, which is based on Open64. The paper describes the existing analyses and optimizations in OpenUH, and explains why the compiler may not apply classical optimizations to OpenMP programs directly. It then presents an enhanced compiler framework including Parallel Control Flow Graph and Concurrent SSA that represent both intra-thread and inter-thread data flow. With this framework, the compiler is able to perform traditional compiler optimizations on OpenMP programs, and it further increases the opportunities for more aggressive optimizations for OpenMP. We describe our current implementation in the OpenUH compiler and use a code example to demonstrate the optimizations enabled by the new framework. This framework may lead to a significant improvement in the performance of the translated code.

1. Introduction

The rapid emergence of multicore and manycore hardware technology has introduced parallelism into the mainstream of software development in all areas. This hardware evolution is a challenge to not only application software vendors, but also to those engaged in language, compiler and system software development. OpenMP (19) is a widely accepted programming model for shared memory systems. One of the major advantages of using OpenMP over a thread library such as pthreads is the ease with which it can be used: a sequential application is parallelized by inserting directives, and runtime library routines, usually with moderate modifications to the program's overall structure. Its incremental parallelization capability provides great flexibility. Its portability and memory model guarantees that OpenMP codes execute correctly on different platforms. The rapid increase in the number of processors/cores in products available in the marketplace requires that OpenMP codes scale well. In the soon-to-be-released OpenMP 3.0, language features, including explicit tasks, have been included to increase OpenMP's applicability and scalability. To ensure the scalability of OpenMP, compiler technology and runtime support are however critical.

OpenMP directives impose a structured programming style and a simple means of synchronization that helps to avoid some types of programming errors. The block structuring of OpenMP code can enable a compiler to analyze this code better than the corresponding threaded code. However, to the best of our knowledge, OpenMP research and commercial compilers do not analyze OpenMP code explicitly, and little or no optimization occurs (25; 16) before the OpenMP code is lowered, when it is translated to threaded C, C++, or Fortran code. At this point, compilers typically perform a restricted set of optimizations and then link the code with a thread library. Some optimizations, such as PRE and code motion, that are widely performed on a sequential program may not be carried out at all on an OpenMP program. This may potentially have a severe impact on the performance of OpenMP codes.

An OpenMP compiler needs to aggressively optimize an OpenMP program to fill the gap between the high level language and increasingly complex parallel architectures. It must be able to apply classical optimizations in addition to invoking much more aggressive optimizations specific to OpenMP. However, some restrictions are required to prevent the use of classical sequential optimizations from changing the semantics of OpenMP programs. (18) listed potential problems caused by conducting classical sequential optimizations, if they are not adapted for parallel programs. Most current OpenMP compilers limit the application of

classical optimizations due to the lack of a framework for inter-thread data flow analysis of OpenMP code. A framework that can represent and analyze OpenMP code with respect to both intra-thread and inter-thread data flow could enable the aggressive compiler optimizations that are critical if we are to further increase OpenMP performance and applicability.

1.1 Overview of OpenUH

The Open64-based OpenUH compiler includes an implementation of OpenMP. This compiler is modularized, with different components that interact via a common IR (WHIRL). Its major functional parts are the three front ends, and the back end, which is further subdivided into the global optimizer, loop nest optimizer, interprocedural analysis/optimizer, and code generator. There are also components for automatic parallelization and for generating source code from the IR, although the resulting source is not guaranteed to be executable. The WHIRL Optimizer (WOPT) module performs data flow analysis at the procedure level and applies various SSA-based optimizations, such as copy propagation, dead code elimination, and partial redundancy elimination (SSAPRE). WOPT is designed such that it can be invoked as a stand-alone optimization phase, referred to as the MAINOPT phase, or in conjunction with another compiler phase, referred to as the PREOPT phase. In the latter mode, WOPT provides analysis and performs some basic optimizations to prepare the code for one of the other compiler components, e.g. the Loop Nest Optimizer (LNO).

In a typical translation of OpenMP code at the O3 optimization level, the compiler front end will generate a high level IR with the OpenMP constructs still intact. Each program unit (e.g. procedure) is then processed by the compiler in turn. First, it is passed to the WOPT, in PREOPT_LNO mode, which prepares the code for the LNO module. After LNO, the compiler lowers the OpenMP constructs to corresponding threaded code (via calls to our OpenMP runtime library). Then, WOPT is again invoked on the translated code in the MAINOPT phase. Here a complete set of data flow optimizations are performed, including SSAPRE. After MAINOPT, the emitted code is in a low level WHIRL format, and the code generator (CG) is invoked.

1.2 OpenMP Programming and Memory Model

OpenMP provides a set of high level directives and runtime interfaces to allow users to easily express the parallelism in their C, C++ and Fortran applications without the need to explicitly create and manage threads. The directives enable the user to specify that a structured block of code (that may span multiple procedures) is to be executed by multiple threads, and to describe how the work inside that parallel region is to be shared among the executing threads. Its directives for expressing worksharing, including *omp for, omp sections, omp single, omp master* are also applied to structured blocks of code. The synchronization directives provided by OpenMP include *omp critical, omp barrier, omp atomic, omp flush, omp ordered*; they are used to protect data and order the execution of operations among threads as required by an algorithm. Data in a parallel region can be set to be shared, private (where each thread has its own copy), firstprivate or lastprivate. The runtime library can be used to dynamically control and query the parallel execution environment. The addition of tasking directives in OpenMP 3.0 will further extend the expressibility and applicability of OpenMP, especially to non-HPC applications.

OpenMP is based on a relaxed consistency memory model that allows each thread to have its own local view of shared data (8) at times during execution: when a synchronization point is reached in the code, consistency is enforced by flushing the values of modified shared data to memory. Most OpenMP worksharing constructs include implicit barriers that ensure the synchronization of thread execution and therefore also serve to keep the data consistent between them. However, the local value of a shared object may or may not be consistent before such a barrier, and OpenMP provides a unique feature with the explicit *omp flush* construct to force the value of one or more shared objects to be written back to memory at an arbitrary point during execution[1]. This allows an aggressive optimizing compiler to safely assume that there are no inter-thread data interactions until a flush operation has been reached. The OpenMP memory model simplifies the compiler analysis for parallel programs because the compiler can perform traditional analysis and optimizations safely between two synchronization operations. However, most compilers perform optimizations after OpenMP has been translated to a threaded code, thereby limiting the sequential analysis and optimizations to code between two synchronizations.

An *omp flush* directive may specify a list of variables that need to be flushed to memory or read from the memory. If the list is empty, all shared variables are flushed. A flush operation executed by a thread means that data associated with the thread is consistent with memory at the point, but the flush operation does not other threads. If two threads need to make their data consistent, then two flush operations must be executed by them: one is to write data to memory, and the other is to read data from memory. The order of the two flush operations needs to be ensured by additional synchronizations. Although most hardware today provides cache coherence mechanisms, flush operations are still necessary to ensure the data coherence with memory. Moreover, OpenMP is a portable programming model, and OpenMP codes should work on machines without cache coherence mechanisms, or potentially even on clusters with a virtual global memory support. In this paper, we assume that OpenMP codes are compliant with the memory model, and are written us-

[1] From the OpenMP 2.5 specification, "This operation makes a thread's temporary view of memory consistent with memory, and enforces an order of the memory operations of the variables explicitly specified or implied."

ing directives to express the parallelism. (In other words, we will assume that the application developer has not relied on thread IDs as a means of distributing work.) The compiler analysis and optimizations described in the paper are based on the OpenMP memory model as described in the current specification, and exploits the structured nature of OpenMP constructs.

1.3 OpenMP Translation

Most OpenMP compilers translate OpenMP into multithreaded code via calls to a custom runtime library either via outlining (2), inlining (16) or an SPMD scheme (9) for clusters. OpenUH/Open64 uses the inlining approach to generate a nested procedure for each parallel region in the original procedure. The advantage of the inlining approach is that the shared variables are visible to the nested procedure by default. A pre-translation phase is carried out first to standardize the use of OpenMP constructs and reduce the number of them that must be subsequently handled by converting *OMP sections* to *OMP do/for*. Then, the compiler translates OpenMP directives to a mixture of suitably modified code and invocations of threaded runtime functions. Since many details of execution, such as the number of iterations in a loop nest that is to be distributed to the threads, and the number of threads that will participate in the work of a parallel region, are often not known in advance, much of the actual work of assigning computations must be performed dynamically. An efficient runtime library to manage program execution is essential.

There are two execution paths in the generated code: one preserves the sequential program, for use when only one thread is used at runtime, and the other is the translated multithreaded code. The rationale for maintaining these two paths is to minimize the potential overhead of translated threaded code when the code is executed sequentially. As we have seen, this does not work out.

Although a precise synchronization-sensitive data flow analysis is undecidable (20) for parallel programs, we believe that a pragmatic approach to analyzing OpenMP programs can be very worthwhile. We have designed and implemented such an analysis in the OpenUH compiler and report upon it in this paper.

We motivate our work in Section 2, where we describe the current OpenUH compiler optimizations and the restrictive manner in which they are applied for parallel program optimizations. We then consider how to enable classical global optimizations to OpenMP programs and describe a data flow analysis framework for OpenMP, along with a pragmatic approach to its implementation. Our Parallel Control Flow Graph and Concurrent SSA is introduced in Section 3. We provide details of our modified Parallel Static Single Assignment analysis in Section 4 and explain our implementation in Section 5. Finally, we discuss related work in Section 6 and give conclusions in Section 7.

2. Motivation for This Work

In order to compare how Open64 performs compiler optimizations for seqential and OpenMP programs, we created a simple test case called *pre-example* to test copy propagation and PRE using the OpenUH/Open64 compiler. Fig. 1(a) shows an OpenMP code fragment that includes a parallel loop. We expected that the performance of this OpenMP code would be comparable to that of the corresponding sequential code when it is executed by only one thread. However, the OpenMP code performed very poorly. The second column of Table 1 shows the execution time when we used the -O3 flag only: in this case, the compiler ignores the OpenMP directives, and compiles and optimizes it as a sequential program. The third column is the execution time obtained when we enabled OpenMP compilation. The performance of pre-example is about 7 times slower than the corresponding sequential code on one thread. In other words, the OpenMP code version exhibited a remarkably poor baseline performance using one thread, when compared with its corresponding sequential code. A performance difference can be seen from some of NAS benchmarks too, especially FT, and UA. Something is wrong here.

We used the source-to-source capability provided by OpenUH/Open64 to view the compiler-translated codes. We found that the compiler performed classical optimizations very well when it ignored the OpenMP directives. The expression $x * y/f1$ is repeatedly executed in the loop and can be hoisted out of it by conducting Partial Redundant Elimination(PRE) and copy propagation optimizations as shown in Fig. 1(b). In this case, the optimized code after PRE is very efficient. However, PRE was simply not applied when the compiler translation of OpenMP directives was enabled. Fig. 1(c) shows the code after this translation: in this case, no optimizations were applied to the code. As a result, the performance was quite different.

The main reason why these optimizations were not applied to the OpenMP code is that the compiler does not have any means to analyze inter-thread data flow. It therefore has to conservatively assume that any shared data may be changed in the parallel region by other threads at any time. A consequence of this is that almost all aggressive optimizations are disabled. Moreover, the optimizations were attempted after the compiler translated the structured OpenMP directives into threaded code, which creates another obstacle for applying these optimizations. Some information provided by OpenMP semantics and structures may be lost after this translation.

Fig. 2 shows an OpenMP code and the compiler-generated threaded code after this translation. Based on OpenMP semantics, it is known in this case that variable *k* has the value 1 after the OpenMP *single* construct. However, in the translated code, a compiler will not be sure if *k* is equal to 1 or not at the *if(k==1)* statement, since the value of *mpsp_status* is unknown at compile time. The real problem here is that com-

```
int main()
{
 double a[N];
 double x = 3.1415;
 double y=3.1415926;
 double f1=2.3132;
 ...
#pragma omp parallel
 {
 #pragma omp single
  {
    x = x * y/f1 * k;
  }
#pragma omp for reduction(+:z)
for(j=0; j<1000; j++)
  for(i=0; i<N; i++)
  {
   a[i]= a[i] * x*y/f1;
   z = z + a[i] + x*y/f1 ;
  }
  printf("results: z= %d\n", z);
 }
}
```

(a) An OpenMP program

```
int main()
{
 double a[N];
 double x = 3.1415;
 double y=3.1415926;
 double f1=2.3132;
 pre_cst = 4.2665196; // x*y/f1
 x = k * pre_cst;
 preg1 = x*3.1415926 / 2.3132;
 for(j0 = 0U; j0 <= (_INT32)(999U);
      j0 = j0 + (_INT32)(1U))
 {
  z_ = 0.0;
  for(i0 = 0U; i0 <= 999999;
      i0 = i0 + (_INT32)(2U))
  {
    a[i0] = (_IEEE64[2])(a[i0]) * preg1;
    z_ = ((_IEEE64[2])(a[i0]) + z_) + preg1;
  }
  z = z + z_;
 }
 printf("results: z= %d\n", z);
}
```

(b) The compiler-optimized code when OpenMP directives are ignored

```
int main()
{
 ....
  __ompc_serialized_parallel();
  x = k * ((x * y) / f1);
  for(j = 0U; j <= (_INT32)(999U);
      j = j + (_INT32)(1U))
  {
    i = 0U;
    i0 = 0U;
    i1 = i0;
    while(i0 <= (_INT32)(999999U))
    {
      a[i0] = a[i0] * ((x * y) / f1);
      z = (a[i0] + z) + ((x * y) / f1);
      i2 = i0 + (_INT32)(1U);
      i0 = i0 + (_INT32)(1U);
      i3 = i0;
    }
    i4 = 1000000U;
  }
  printf("results: z= %d\n", z);
}
```

(c) The non-optimized code when OpenMP directives are translated

Figure 1. An OpenMP code compiled with/without -mp flag

compiler flags used / execution time (seconds)	-O3	-O3 -mp -gnu3
pre-example	7.42	46.8
NAS benchmark FT: CLASS=A	18.45	26.17
NAS benchmark UA: CALSS=A	130.31	220.15

Table 1. Performance comparison: sequential vs. OpenMP program on one thread

piler optimizations for OpenMP programs are performed after the OpenMP code has been translated into threaded code; further, the subsequent optimizations are still based on the standard sequential control flow graph, which does not provide sufficient information to enable them. Note, too, that the translation has introduced many "if" conditions along with the runtime functions, which will typically force the compiler to make conservative assumptions.

Another challenge to the optimization of OpenMP is that the translation process introduces additional pointers. During lowering, a parallel region will be converted into a procedure and shared variables are passed to it by providing their addresses as parameters. Inside the procedure, the shared variables are accessed by pointers, rather than via explicit references. Without an accurate pointer analysis, this will further limit the compiler's ability to conduct optimizations.

This overall behavior is unfortunately typical of commercial compilers, whose translation of OpenMP is fairly similar to our own. The Intel compiler is able to conduct some optimizations for translated OpenMP code, but the optimizations are limited to *T-regions* only (25), which is an OpenMP parallel construct. We believe that an OpenMP compiler should attempt to apply optimizations to OpenMP code *before* its constructs are translated. In order to do so, the compiler will require a framework that represents the control flow of the OpenMP program. In the following sections, we describe how we have worked to provide just such a framework in OpenUH.

3. OpenUH Optimizations

OpenUH uses the WHIRL Optimizer (WOPT) at one or more phases during the compilation process (depending on the level of optimization) to globally analyze and optimize the code at a procedural level. Fig.3(a) illustrates how WOPT can be invoked in various contexts. In addition to restructuring the code in preparation for subsequent compiler phases, it can also compute and export def-use and alias information. In a typical compilation of an OpenMP program at the -O3 optimization level, the compiler will invoke WOPT just before LNO for def-use information as well as to canonicalize loop induction variables. This is called the PREOPT phase of the Optimizer. The presence of parallel regions hinders some optimizations in PREOPT. Specifically, because the optimizer lacks parallel data flow analyses, it conservatively restricts optimizations involving potentially shared variables. Before PREOPT, the compiler performs a OpenMP pre-lowering phase that inserts memory barrier function calls around OpenMP synchronization constructs. These memory barriers also restrict optimizations to be performed across them.

Subsequently, after OpenMP translation WOPT is again invoked on the program unit in what is called the MAINOPT phase. MAINOPT will also be used for microtasks generated during OpenMP translation, but as we will see there is typically very limited scope for optimization. In addition to repeating the analyses and optimizations carried out earlier in PREOPT, MAINOPT will also perform IR lowering, execute a comprehensive set of optimizations via SSA Partial

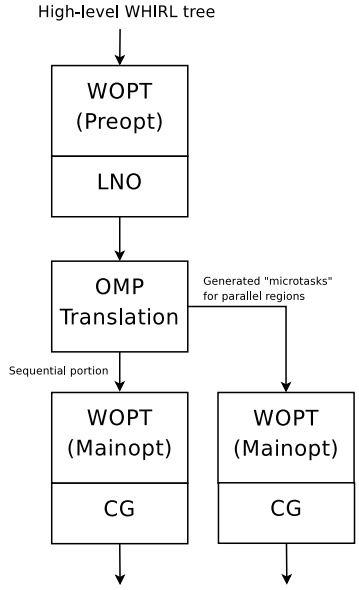
```
#pragma omp single
  {
    k = 1;
  }
  if(k==1) ...
```

(a) An OpenMP program with single construct

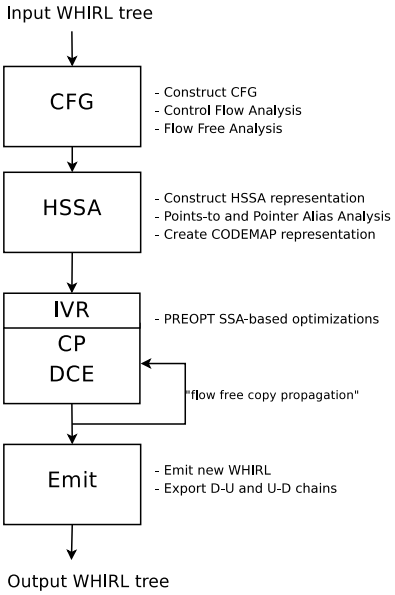
```
mpsp_status = __ompc_single(__ompv_temp_gtid);
if(mpsp_status == 1)
{
  k = 1;
}
__ompc_end_single(__ompv_temp_gtid);
if(k==1) ...
```

(b) The corresponding compiler translated threaded code

Figure 2. A compiler translated OpenMP code



(a) OpenMP Compilation



(b) PREOPT Phase

Figure 3. OpenUH WOPT

Redundancy Elimination (SSAPRE) (10), and export alias information for CG. We will now detail the specific analyses and optimizations provided by WOPT and discuss how the presence of OpenMP is taken into account during PREOPT.

3.1 Control Flow and Flow-Free Alias Analysis

The major functionality provided by WOPT in PREOPT phase is depicted in Fig. 3(b). WOPT traverses the input tree and creates a special auxiliary symbol table for its own use. This pass also entails certain optimizations such as folding of indirect loads, but parallel regions are left intact. It then traverses the input WHIRL tree and performs alias classification for memory operations based on a nearly linear time points-to-analysis (24). A control flow graph (CFG) is created from the input tree, upon which various preliminary control flow analyses are performed. While constructing the CFG WOPT will make note of the various parallel region constructs it encounters, including standard parallel regions (*$omp parallel*) and parallel loops (*$omp parallel do*). The control flow analyses include the creation of the dominator and postdominator tree, calculating the dominance frontiers for each node of the CFG, and analyzing the loop structures. WOPT will then performs a flow free alias analysis (FFA) on the CFG, where the code is annotated with aliasing information in the form of μ and χ lists. μ- and χ-functions are used to represent hidden defs and uses of scalars, where μ-functions represent MayUses and χ-functions represent MayDefs. For a given statement, these describe any potentially implicit uses or definitions of variables. A first pass will generate μ and χ lists for scalar variables and map occurrences of indirect loads and stores to virtual variables. A second pass will then insert the virtual variables into the μ and χ lists.

3.2 HSSA

The control flow analysis and FFA are used to construct a Hashed Static Single Assignment (HSSA) form for the code. HSSA is an extension of the standard SSA form and enables the SSA representation of both scalar variables and indirect memory operations. In the standard SSA form, ϕ-functions

are inserted to represent scalar variables that could potentially have more than one value. HSSA extends this concept by also considering the μ- and χ-functions described above, aliasing results in many more potential reaching definitions at a given node in the CFG. A flow sensitive alias analysis is carried out on the CFG in reverse dominator tree order, followed by dead store elimination. In codes where aliasing is prevalent,"zero versioning" is used to reduce the number of versions for variables. A unique value number is assigned to live scalar and virtual variable versions using a hash table (only one number needed for zero versions of a given variable). Then, WOPT will carry out a preorder traversal of the dominator tree and assign value numbers to expressions as well. This will allow WOPT to determine the equivalence of two expressions as well as scalars and virtual variables (5). During this process, if a parallel region exists in the program unit, then any *LHS* variables that are not private inside an enclosing parallel region will be marked as *shared*.

3.3 PREOPT Optimizations

WOPT then carries out various optimizations, most notably induction variable recognition (IVR), copy propagation (CP), and dead code elimination (DCE). In IVR, induction variables are identified for all loops, where one IV that is incremented by 1 for each iteration is designated as primary, and the rest as secondary. Then, all secondary IVs are redefined in terms of the primary IV at the start of the loop body. Further, an assignment statment for each IV and its respective exit value is inserted after the loop. IVR together with the code restructuring provided by subsequent optimizations steps will effectively canonicalize the loop induction variables and prepare the code for LNO (17). When processing loops in IVR, loops that in turn contain parallel loops are skipped. Parallel loops that are not enclosed by an outer loop can optionally be processed, however, though reduction variables will not be recognized as an IV. When processing parallel loops, IVR will first preprocess the OpenMP pragma list (e.g. shared, private, reduction, etc.), perform the conversion of induction variables, and then update the pragma list.

CP is a generalized form of constnat propagation which includes the propagation of constants and expressions, as well as expression simplification. WOPT conservatively restricts the propagation of values defined by variables with the *shared* attribute due to a lack of data flow analysis for shared variables. By propagating the calculation of expressions, there is the risk of a performance hit. This issue can be dealt with during the following DCE step, where both unreachable code and dead stores are eliminated. A further restrict during DCE is that scalar stores in parallel regions are not eligible for elimination. WOPT will next perform a flow free copy propagation where the above optimizations are repeated multiple times until it determines that it can not eliminate any more dead code. The same restrictions when deal with parallel regions and shared variables continue to apply. This is the extent of the optimizations carried out by WOPT during PREOPT. PREOPT concludes by generating the new optimized WHIRL tree from the optimized SSA form and exporting def-use information that the LNO needs to construct its dependence graph.

3.4 MAINOPT Optimizations

During MAINOPT, WOPT will have to prepare the code for code generation. Before constructing the auxiliary symbol table, it will perform IR lowering on several types of WHIRL nodes, including complex numbers, arrays, region exits, and more. Then, WOPT will behave much as it did during PREOPT, except that in the absence of OpenMP parallel regions, it can more aggressively optimize variables in sequential regions that were previously considered shared. After flow free copy propagation, MAINOPT will remove critical edges from the CFG and then perform SSAPRE. Within the SSAPRE framework (10), MAINOPT can perform many of important optimizations including common subexpression, strength reduction, code hoisting, linear function test replacement, and register promotion. It performs a full redundancy elimination based on value numbering (VNFRE), followed by PRE for expressions (EPRE) and partial DSE. Next it will execute local register promotion and PRE for loads and stores (LPRE and SPRE), within which WOPT can perform the first phase of its register promotion. Next WOPT will execute a bit-wise variant of DCE. To finalize, it emits a lower level, optimized WHIRL tree, and analyzes this tree to promote scalar variables to registers (RVI) in preparation for CG.

4. Parallel Data Flow Analysis Framework

4.1 Parallel Control Flow Graph

We have designed a Parallel Control Flow Graph (PCFG) for representing OpenMP programs in order to enable aggressive optimizations such as those described in the previous section, while guaranteeing correctness. The PCFG is not unlike the Program Execution Graph (1) and the Synchronized Control Flow Graph (4), proposed by other researchers. The distinction between our PCFG and their flowgraphs is that our PCFG is based upon the relaxed memory model of OpenMP, and its barrier and flush synchronizations, instead of event-based synchronizations (such as signal-wait).

4.1.1 A Definition of The Parallel Control Flow Graph

The PCFG is a directed graph (N, E, s, e), where the set of nodes, N, includes basic nodes, composite nodes, super nodes, and barrier nodes as defined below; E is a set of directed edges including sequential edges, parallel edges and conflict edges; s and e represent the entry and exit of a parallel region, respectively.

Definition 1. ***Basic node****: a basic node is a basic block, i.e. a code segment with only one entry point and one exit point; an* ***omp flush*** *directive is considered to be a basic node.*

Many OpenMP constructs contain implicit flush operations. These operations are made explicit before the PCFG creation.

Definition 2. ***Composite node****: a composite node is composed of an OpenMP worksharing or synchronization construct and the basic nodes associated with it, or of consecutive basic nodes executed by all threads in a parallel region that neither contain nor are immediately enclosed in an OpenMP construct (other than a parallel construct). These OpenMP constructs are* omp master, omp single, omp sections, omp for, omp worksharing (Fortran), omp critical, omp atomic.

A composite node may contain nested OpenMP constructs following OpenMP semantics. For example, there may be an *omp critical* inside an *omp for* construct. In this case, the composite node for *omp critical* is nested inside the composite node of *omp for*.

Definition 3. ***Supernode****: a super node consists of all composite nodes between two barriers.*

Supernodes are separated by barriers, and they are executed in the order of "happens before". All shared variables and control flow are synchronized at the barriers. The data flow between supernodes follows sequential order, and can be analyzed safely by traditional algorithms. The parallel data flow analysis is performed within each supernode.

Definition 4. ***Barrier node****: a barrier node contains an* ***omp barrier*** *directive only. It is used to synchronize data and threads' execution. In our PCFG, it specifies the sequential order in which supernodes are executed, and reflects the data synchronizations. Both explicit and implicit barriers are represented by a barrier node.*

To represent the intra-thread and inter-thread data flow in OpenMP programs, we need to define the following three kinds of edges.

Definition 5. ***Sequential edge****: a sequential edge indicates a control flow path that is executed by a single thread or by all threads. If a path is executed by all threads, the path is connected by sequential edges too. It is used to represent the data flow propagation order among different nodes for an individual thread.*

A control flow edge in the sequential part of an OpenMP program, which includes the code outside a parallel region, is a sequential edge. Control flow edges representing flow of code within worksharing and synchronization constructs (except flush) inside a parallel region are also sequential edges. In the latter case, this is because the code is executed by either one thread, or all threads.

Definition 6. ***Parallel edge****: a parallel edge is used to represent a conditional branch where different threads may take different paths based on the value evaluated on the branch.*

Parallel edges are used at a conditional branch to connect to different composite nodes inside a supernode. When the corresponding condition is evaluated, the results obtained by individual threads may differ. In this case, different threads will execute different composite nodes. OpenMP constructs such as OMP sections, OMP single and OMP master have implicit conditional branches to allow different threads to take different paths. These constructs are connected in our PCFG with parallel edges. The difference between a branch represented by sequential edges and a branch represented by parallel edges is that a parallel edge must be executed by at least one thread, and the sequential edge may or may not be executed depending on the condition of the branch.

Definition 7. ***Conflict edge****: there is a conflict if two different threads access the same memory location and at least one of them updates it. A conflict edge indicates inter-thread data propagation; it connects two flush operations inside a supernode that have at least one common variable in their flush variables list.*

The OpenMP memory model permits data to be communicated between different threads via flush operations. The conflict edge can be bidirectional and unidirectional based on a read or write of a memory access (DU/UD/DD) between different threads along with the edge. If one thread writes a variable and another thread reads the same variable, then the conflict edge is a unidirectional edge pointing from the write flush to the read flush operation. If both threads perform reads and writes to different variables, then the conflict edge becomes bidirectional. Please note that the conflict edge only indicates the data flow between different threads; due to the nondeterministism of parallel programs, such a data transfer may or may not happen. A compiler, however, has to conservatively assume that data may flow between different threads along all of the conflict edges.

4.1.2 Parallel Control Flow Graph Creation

Fig. 4 shows how composite nodes representing worksharing constructs are connected by parallel and sequential edges. In Fig. 4 A and B, different threads may take different paths, so that parallel edges represent the branches for different threads. In Fig. 4 C, the *omp for* loop will be executed by all threads, and we use a sequential edge to connect it. Based on the *omp for* directive's semantics, the enclosed loop should not have any data dependence. We treat the loop as a sequential loop in the PCFG. Fig. 4 D presents the PCFG for the *omp critical* construct. A critical section is executed by multiple threads one by one, but never at the same time. A flush operation exists at the entry to and exit from a critical region. We create a conflict edge that connects the flush operation at the exit to the one at the entry of the critical

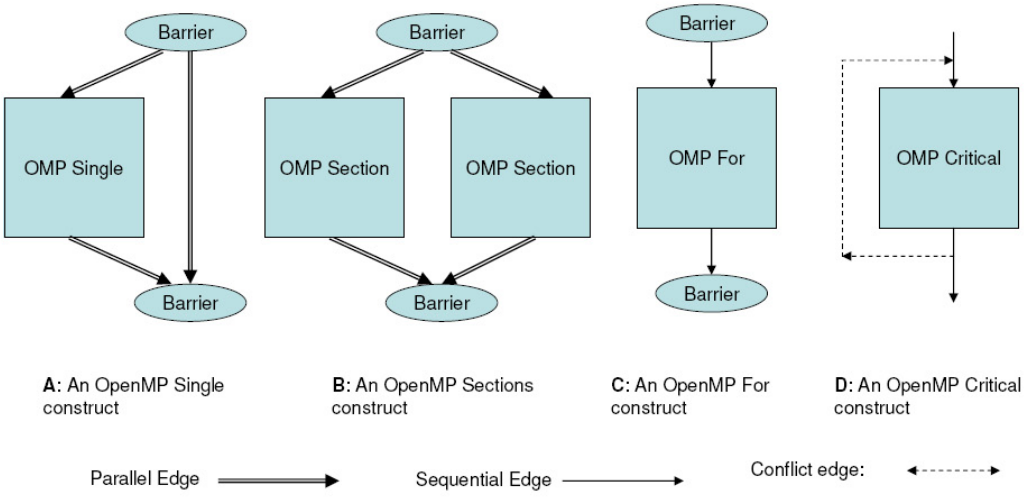


Figure 4. OpenMP Worksharing Constructs in PCFG

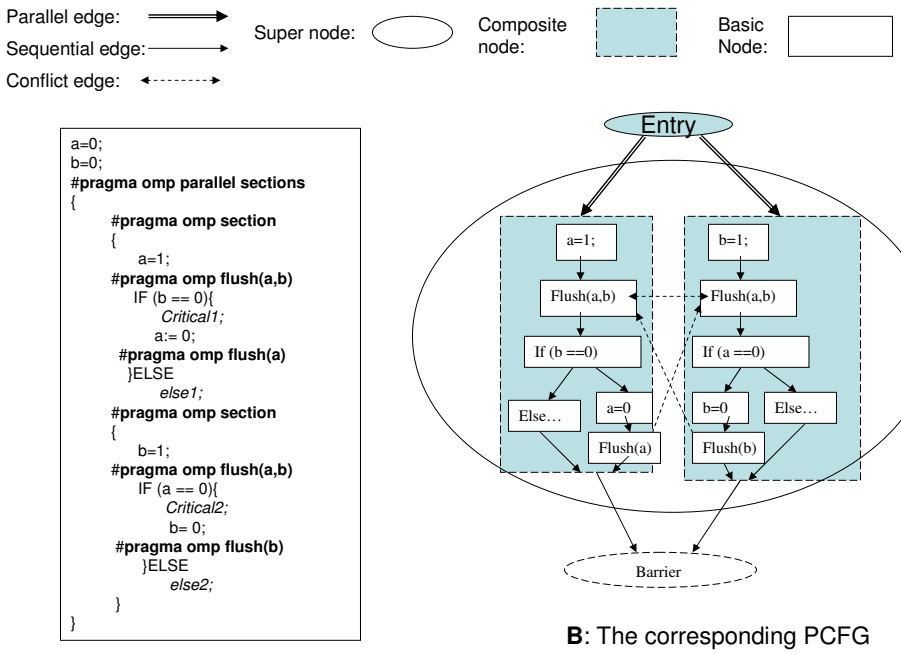


Figure 5. An OpenMP Section example and its PCFG

region to represent the fact that data may propagate to the next thread that executes the critical region.

4.2 An Example of a Parallel Control Flow Graph

Fig. 5 gives a code fragment that is similar to code used by Vollmer in (26), and converted into an OpenMP code. It is intended to implement a simple synchronization scheme us-

ing *a* and *b* to prevent *critical1* and *critical2* from executing at same time. Vollmer claims that a compiler may generate incorrect code by applying sequential data flow analysis to this. However, it is legal to perform optimizations on the two *omp sections* in the OpenMP code if there is no *omp flush* included in either of the two sections, since the definitions of shared variables *a* and *b* may or may not be visible by the other thread until the barrier is reached.

We modified the code by adding four *omp flush* operations to it: it is then no longer legal to perform sequential optimizations, since the definitions are required to be visible to other threads at the flush point. Note that the *flush(a,b)* operation is necessary and compliant with the OpenMP standard to flush variables both *a* and *b*, since it prevents any out of order execution of the writes to *a* and *b* and the following *if* condition in a single thread. Conflict edges represent the possible data propagation between threads. We connect flushes based on the write and read (DU/UD/DD) for the flushed variables. This inter-thread data flow is not represented by a sequential control flow graph. Our PCFG represents multiple threads' execution paths that provides a framework for compiler to conduct optimizations correctly for OpenMP programs.

The OpenMP flush operation implies that the program writes and reads the shared data into and from the main memory. However, the operation does not guarantee that data is consistent at the flush point. In our PCFG, we use the conflict edge to represent data communications, but these communications are not guaranteed in general.

4.3 Concurrent Static Single Assignment

The existing OpenUH HSSA cannot be used with the previously introduced PCFG. It also does not take into account possible conflicts that may result from inappropriate synchronization. We introduce a new modified SSA form, called Concurrent Static Single Assignment (CSSA) to complement the PCFG that we propose for OpenUH. In addition to the ϕ functions used in HSSA, CSSA will include ψ- and π-functions that are similar to those proposed by Lee *et al.* (15). In CSSA, the ϕ-functions $\phi(v1, v2, ..., vn)$ are placed at the end of joins following the control flow graph where multiple values of the same variable are possible. In parallel programs, the ψ-functions $\psi(v1, v2, ..., vn)$ are placed, instead of ϕ-functions, at the end of parallel regions where multiple values of the same variable are possible. π-functions $\pi(v1, v2, ..., vn)$ are placed in the PCFG where multiple values of a shared variable may have different values from multiple threads, potentially causing a conflict.

Fig. 6 shows the PCFG and CSSA for the example in Fig 5. The CSSA is created by first computing the execution order of basic blocks with synchronizations. Next, ϕ-functions are inserted at each joined point to represent the creation of a new variable based on one of the possible definitions reach that point. The ψ-functions are inserted at the end of each supernode to represent one of the definitions in different threads for a shared variable in the supernode. Variables are then renamed to ensure single assignment. Finally, the conflict edges of the PCFG are used to determine where π-functions are placed to represent the possible definitions of a variable from multiple threads.

5. Extending Global Optimizer for OpenMP Analysis

In this section, we discuss how we have integrated the parallel data flow framework described above into Open64's global optimizer (WOPT). We have extended the PREOPT phase of WOPT that is performed before LNO to analyze and optimize OpenMP codes. This entails extending the existing control flow analysis to take into account *parallel* and *conflict* edges in the PCFG described above and reformulating the HSSA representation to take into account the new use-def information. From this point, we can enable optimizations that were previously restricted to non-shared variables such as copy propagation, as well as enable more aggressive optimizations within parallel regions, such as dead store elimination (DSE) and expression PRE (EPRE).

5.1 PCFG Construction

We have extended the CFG created in PREOPT to include extra control flow and data flow information for parallel regions. As before, PREOPT will create CFG so that basic blocks connected by control dependence edges are formed over a given procedure. At this point, we add a *parallel control flow analysis* step which converts the existing CFG into a PCFG. Instead of fundamentally restructuring the CFG, our approach is to add annotations to parallel regions which will allow the optimizer to group basic blocks into composite nodes, and in turn group composite nodes into supernodes. We must also add *sequential*, *parallel*, and *conflict* edges, as appropriate, to the parallel regions. Whereas originally the CFG connects basic blocks together, parallel edges are used to connect composite nodes.

In order to create a PCFG, the following steps are followed. The approach our algorithm uses is to first identify OpenMP constructs, as well as explicit and implicit barriers and flushes. These are then used to construct both supernodes and composite nodes. Basic nodes are

1. Construction of the PCFG begins by making barriers explicit. To accomplish this, all OpenMP worksharing directives without a *nowait* clause are replaced by the same OpenMP construct with a *nowait* clause followed by an explicit barrier. (Note that this does not apply to an *omp master*, which does not imply a barrier.)
2. We then make implicit flush operations explicit. For example, *omp critical* and *omp order* have a flush operation at both the beginning and the end of the construct.

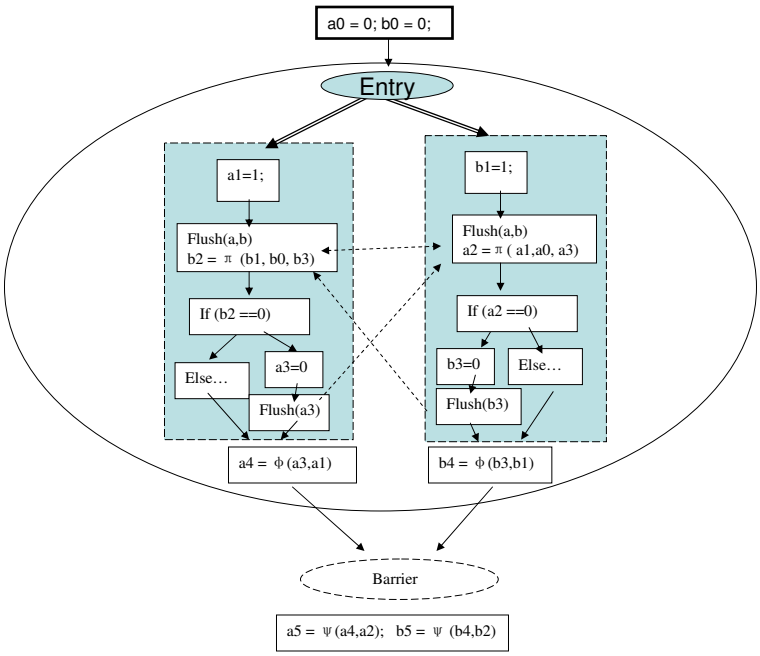


Figure 6. An example of PCFG and CSSA

3. We then start creating the PCFG from the entry to a parallel region, by forming the basic nodes. Basic nodes are connected via sequential edges.
4. All basic nodes are included in a composite node until an OpenMP construct is reached. The OpenMP construct and its enclosed basic nodes also form a composite node. If a worksharing directive including *omp single*, *omp master*, and *omp sections* is reached, parallel edges are used to connect the composite node and its successor composite node.
5. When a barrier is encountered, a supernode is created to include all composite nodes between two consecutive barriers. A barrier node contains only the barrier. A supernode is connected to a barrier node via a sequential edge.

Once this has been created, we are able to perform control flow analyses over the new PCFG. This requires modifying the existing algorithms for calculating dominator and post-dominator trees and dominance frontier to take into account parallel edges. CSSA can be implemented within OpenUH by modifying the existing ϕ-function algorithm to allow the representation of ψ- and π-functions. These operators can be easily added to existing ϕ nodes, while preserving the single assignment property. We are exploring how to represent ψ- and π-functions in terms of ϕ-functions, thereby simplifying our implementation and enabling the use of existing OpenUH optimization algorithms that are currently based on ϕ-functions.

5.2 Results

With the PDFA framework introduced in this paper, a compiler can perform optimizations before OpenMP constructs are lowered to threaded code, and remove the conservative restrictions on optimizating in the presence of shared data. Although the implementation of our PDFA framework is still ongoing, an early evaluation showed that the *pre-example* code of Fig.1 has achieved a performance similar to its corresponding sequential version. The translation from IR to source indicates that the translated code is optimized with copy propagation and PRE as we would expect in a sequential code.

6. Related Work

There have been a number of efforts that develop parallel dataflow analysis. Shasha and Snir (22) showed that a parallel program may violate sequential consistency if $E \bigcup P$ contains a cycle, where E is the execution order and P is the order of variable accesses. Based on Shash and Snir's work, Krishnamurthy and Yelick (12) proposed a compiler framework to analyze parallel programs with explicit barriers, post-wait and lock synchronization. Knoop *et. al* (11)

has developed a theory of Parallel Data Flow Analysis and proved that it is possible to perform it for parallel programs (26). Grunwald et. al (7) presents data flow equations for explicit parallel programs, but this work analyzes parallel sections only. Other work (23; 15) use the SSA form to solve data flow problems for parallel programs. However, this research does not target OpenMP programs.

To the best of our knowledge, OpenMP compiler optimization is constrained to the code within an OpenMP parallel construct. Global optimizations and code motions are not performed across parallel constructs (25). Satoh *et. al* (21) has presented a Parallel Flow Graph and compiler analysis for OpenMP programs. However, the paper does not cover all OpenMP constructs and does not consider the impact of flush operations. Yet the flush operation is key to ensuring that any optimizations performed do not violate OpenMP memory consistency rules, while it also prevents aggressive global optimizations. Zhang *et al.* (27) presented a concurrency analysis for OpenMP, but the work was focused on matching barrier synchronizations, instead of optimizations. Our early ideas on the design of a parallel dataflow framework for OpenMP were described in (13).

7. Conclusions and Future Work

This paper describes a compiler framework that enables high-level data flow analysis and optimizations for OpenMP. The framework represents the intra- and inter-thread data flow in OpenMP programs based on the relaxed memory model. It creates and/or increases the opportunities for performing a range of traditional global optimizations on OpenMP code before it is lowered to explicitly threaded code. By taking OpenMP semantics into consideration, it also enables more aggressive optimizations that are specific to this programming interface. We are currently implementing this framework in the OpenUH compiler (16). We plan to evaluate our work using more sophisticated benchmarks and also intend to investigate how it may support the implementation of OpenMP on clusters (9), as well as to help statically detect race conditions in an OpenMP program. We believe that this framework will improve overall OpenMP program performance and support our long-term goal of providing a highly productive, effective parallel programming model for a wide range of shared memory platforms.

References

[1] V. Balasundaram and K. Kennedy. Compile-time detection of race conditions in a parallel program. In *ICS '89: Proceedings of the 3rd international conference on Supercomputing*, pages 175–185, Crete, Greece, June 1989. ACM Press.

[2] C. Brunschen and M. Brorsson. OdinMP/CCp - a portable implementation of OpenMP for C. *Concurrency - Practice and Experience*, 12(12):1193–1203, 2000.

[3] D. Buttlar, B. Nichols, and J. P. Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., 1996.

[4] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 21–30, Seattle, Washington, United States, March 1990. ACM Press.

[5] F. C. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In *Computational Complexity*, pages 253–267, 1996.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[7] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 159–168, San Diego, California, United States, July 1993. ACM Press.

[8] J. P. Hoeflinger and B. R. de Supinski. The openmp memory model. In *the 1st International Workshop on OpenMP (IWOMP 2005)*, Eugene, Oregon, United States, June 2005.

[9] L. Huang, B. Chapman, and Z. Liu. Towards a more efficient implementation of OpenMP for clusters via translation to Global Arrays. *Parallel Computing*, 31(10-12), 2005.

[10] R. Kennedy, S. Chan, S.-M. Liu, R. Lo, P. Tu, and F. Chow. Partial redundancy elimination in SSA form. *ACM Trans. Program. Lang. Syst.*, 21(3):627–676, 1999.

[11] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Program. Lang. Syst.*, 18(3):268–299, 1996.

[12] A. Krishnamurthy and K. A. Yelick. Optimizing parallel programs with explicit synchronization. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–204, La Jolla, California, United States, June 1995.

[13] G. S. L. Huang and B. Chapman. Parallel data flow analysis for openmp programs. In *International Workshop on OpenMP (IWOMP 2007)*, Beijing, China, June 2007.

[14] J. Lee, S. P. Midkiff, and D. A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Languages and Compilers for Parallel Computing*, pages 114–130, 1997.

[15] J. Lee, D. A. Padua, and S. P. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 1–12, Atlanta, Georgia, United States, August 1999. ACM SIGPLAN.

[16] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. OpenUH: An optimizing, portable OpenMP compiler. *Concurrency and Computation: Practice and Experience, Special Issue on CPC'2006 selected papers*, pages 2317–2332, December 2006.

[17] S.-M. Liu, R. Lo, and F. Chow. Loop induction variable canonicalization in parallelizing compilers. In *PACT '96: Proceedings of the 1996 Conference on Parallel Architectures and*

Compilation Techniques, page 228, Washington, DC, USA, 1996. IEEE Computer Society.

[18] S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. In *1990 International Conference on Parallel Processing*, volume II, pages 105–113, St. Charles, Ill., 1990.

[19] OpenMP: Simple, portable, scalable SMP programming. http://www.openmp.org, 2006.

[20] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.

[21] S. Satoh, K. Kusano, and M. Sato. Compiler optimization techniques for OpenMP programs. *Scientific Programming, Special Issue: OpenMP*, 9(2,3):131–142, 2001.

[22] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

[23] H. Srinivasan, J. Hook, and M. Wolfe. Static single assignment for explicitly parallel programs. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 260–272, Charleston, South Carolina, United States, January 1993. ACM Press.

[24] B. Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM.

[25] X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito, and E. Su. Intel OpenMP C++/Fortran compiler for hyper-threading technology: Implementation and performance. *Intel Technology Journal*, 6:36–46, 2002.

[26] J. Vollmer. Data flow analysis of parallel programs. In *PACT '95: Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, pages 168–177, Manchester, United Kingdom, 1995. IFIP Working Group on Algol.

[27] Y. Zhang, E. Duesterwald, and G. R. Gao. Concurrency analysis for shared memory programs with textually unaligned barriers. In *the 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2007)*, Illinois, USA, October 2007.