# OpenUH: An Optimizing, Portable OpenMP Compiler

Chunhua Liao[1], Oscar Hernandez[1], Barbara Chapman[1], Wenguang Chen[2], and Weimin Zheng[2]

[1] Computer Science Department, University of Houston, USA
{liaoch, oscar, chapman}@cs.uh.edu
[2] Computer Science Department, Tsinghua University, China
{cwg, zwm-dcs}@tsinghua.edu.cn

**Abstract.** OpenMP has gained wide popularity as an API for parallel programming on shared memory and distributed shared memory platforms. Despite its broad availability, there remains a need for a portable, robust, open source, optimizing OpenMP compiler for C/C++/Fortran 90, especially for teaching and research, e.g. into its use on new target architectures, such as SMPs with chip multithreading, as well as learning how to translate for clusters of SMPs. In this paper, we present our efforts to design and implement such an OpenMP compiler on top of Open64, an open source compiler framework, by extending its existing analysis and optimization and adopting a source-to-source translator approach where a native back end is not available. The compilation strategy we have adopted and the corresponding runtime support are described. The OpenMP validation suite is used to determine the correctness of the translation. The compiler's behavior is evaluated using benchmark tests from the EPCC microbenchmarks and the NAS parallel benchmark.

## 1 Introduction

OpenMP [1], a set of compiler directives and runtime library routines, is the de-facto programming standard for parallel programming in C/C++ and Fortran on shared memory and distributed shared memory systems. Its popularity stems from its ease of use, incremental parallelism, performance portability and wide availability. Recent research at language and compiler levels, including our own, has considered how to expand the set of target architectures to include recent system configurations, such as SMPs based on Chip Multithreading processors [2], as well as clusters of SMPs [3]. However, in order to carry out such work, a suitable compiler infrastructure must be available. In order for application developers to be able to explore OpenMP on the system of their choice, a freely available, portable implementation would be desirable.

Many compilers support OpenMP today, including such proprietary products as the Intel Linux compiler suite, Sun One Studio, and SGI MIPSpro compilers. However, their source code is mostly inaccessible to researchers and they cannot be used to gain an understanding of OpenMP compiler technology or

to explore possible improvements to it. Several open source research compilers (Omni OpenMP compiler [4], OdinMP/CCp [5], and PCOMP [6]) are available. But none of them translate all of the source languages that OpenMP supports, and one of them is a partial implementation only. Therefore, there remains a need for a portable, robust, open source and optimizing OpenMP compiler for C/C++/Fortran 90, especially for teaching and research into the API.

In this paper, we describe the design, implementation and evaluation of OpenUH, a portable OpenMP compiler based on the Open64 compiler infrastructure with a unique hybrid design that combines a state-of-the-art optimizing infrastructure with a source-to-source approach. OpenUH is open source, supports C/C++/Fortran 90, includes numerous analysis and optimization components, and is a complete implementation of OpenMP 2.5. We hope this compiler (which is available at [7]) will complement the existing OpenMP compilers and offer a further attractive choice to OpenMP developers, researchers and users.

The reminder of this paper is organized as follows. Section 2 describes the design of our compiler. Section 3 presents details of the OpenMP implementation, the runtime support as well as the IR-to-source translation. The evaluation of the compiler is discussed in Section 4. Section 5 reviews related work and the concluding remarks are given in Section 6 along with future work.

## 2 The Design of OpenUH

Building a basic compiler for OpenMP is not very difficult since the fundamental transformation from OpenMP to multithreaded code is straightforward and there are already some open source implementations that may serve as references. However, it is quite a challenge to build a complete, robust implementation which can handle real applications. But such a compiler is indispensable for real-world experiments with OpenMP, such as considering how a new language feature or an alternative translation approach will affect the execution behavior of a variety of important codes. Given the exceptionally high cost of designing this kind of compiler from scratch, we searched for an existing open-source compiler framework that met our requirements.

We chose to base our efforts on the Open64 [8] compiler suite, which we judged to be more suitable than, in particular, the GNU Compiler Collection [9]. Open64 was open sourced by Silicon Graphics Inc. from its SGI Pro64 compiler targeting MIPS and Itanium processors. It is now mostly maintained by Intel under the name Open Research Compiler (ORC) [10], which targets Itanium platforms. Several other branches of Open64, including our own, have been created to translate language extensions or perform research into one or more compilation phases. For instance, the Berkeley UPC compiler [11], extends Open64 to implement UPC [12]. Open64 is a well-written, modularized, robust, state-of-the-art compiler with support for C/C++ and Fortran 77/90. The major modules of Open64 are the multiple language frontends, the interprocedural analyzer (IPA) and the middle end/back end, which is further subdivided into the loop nest optimizer (LNO), global optimizer (WOPT), and code generator (CG).

Five levels of a tree-based intermediate representations (IR) called WHIRL exist in Open64 to facilitate the implementation of different analysis and optimization phases. They are classified as being Very High, High, Mid, Low, and Very Low levels, respectively. Most compiler optimizations are implemented on a specific level of WHIRL. For example, IPA and LNO are applied to High level WHIRL while WOPT operates on Mid level WHIRL. Two internal WHIRL tools were embedded in Open64 to support the compiler developer; one was *whirlb2a*, used to convert whirl binary dump files into ASCII format, and the other was *whirl2c/whirl2f*, to translate Very High and High level WHIRL IR back to C or Fortran source code. However, the resulting output code was not compilable.

The original Open64 included an incomplete implementation of the OpenMP 1.0 specification, inherited from SGI's Pro64 compiler. Its legacy OpenMP code was able to handle Fortran 77/90 code with some OpenMP features until the linking phase. The C/C++ frontend of Open64 was taken from GCC 2.96 and thus could not parse OpenMP directives. Meanwhile, there was no corresponding OpenMP runtime library released with Open64. A separate problem of Open64 was its lack of code generators for machines other than Itaniums. One of the branches of Open64, the ORC-OpenMP [13] compiler from Tsinghua University that was worked on by two of the authors of this paper, tackled some of these problems by extending Open64's C frontend to parse OpenMP constructs and by providing a tentative runtime library. Another branch working on this problem was the Open64.UH compiler effort at the University of Houston, worked on by the remainings authors of this paper. It focused on the pre-translation and OpenMP translation phases. A merge of these two efforts has resulted in the OpenUH compiler and associated Tsinghua runtime library. More recently, a commercial product based on Open64 and targeting the AMD x8664, the Pathscale EKO compiler suite [14], was released with support for OpenMP 2.0.

The Open64.UH compiler effort designed a hybrid compiler with object code generation on Itaniums and source-to-source OpenMP translation on other platforms. The OpenUH compiler described in this paper uses this design, exploits improvements to Open64 from several sources and relies on an enhanced version of the Tsinghua runtime library to support the translation process. It aims to preserve most optimizations on all platforms by recreating compilable source code right before the code generation phase.

Fig. 1 depicts an overview of the design of OpenUH. It consists of the frontends, optimization modules, OpenMP transformation module, a portable OpenMP runtime library, a code generator and IR-to-source tools. Most of these modules are derived from the corresponding original Open64 module. It is a complete compiler for Itanium platforms, for which object code is produced, and may be used as a source-to-source compiler for non-Itanium machines using the IR-to-source tools. The translation of a submitted OpenMP program works as follows: first, the source code is parsed by the appropriate extended language frontend and translated into WHIRL IR with OpenMP pragmas. The next phase, the interprocedural analyzer (IPA), is enabled if desired to carry out interprocedural alias analysis, array section analysis, inlining, dead function

and variable elimination, interprocedural constant propagation and more. After that, the loop nest optimizer (LNO) will perform many standard loop analyses and optimizations, such as dependence analysis, register/cache blocking (tiling), loop fission and fusion, unrolling, automatic prefetching, and array padding. The

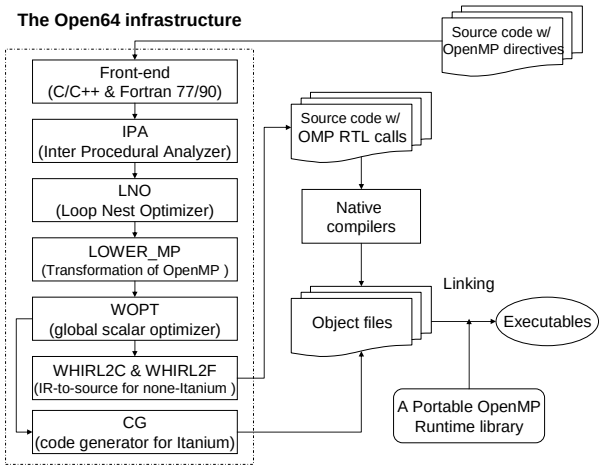


**Fig. 1.** OpenUH: an optimizing and portable OpenMP compiler based on Open64

transformation of OpenMP, which lowers WHIRL with OpenMP pragmas into WHIRL representing multithreaded code with OpenMP runtime library calls, is performed after LNO. The global scalar optimizer (WOPT) is subsequently invoked. It transforms WHIRL into an SSA form for more efficient analysis and optimizations and converts the SSA form back to WHIRL after the work has been done. A lot of standard compiler passes are carried out in WOPT, including control flow analysis (computing dominance, detecting loops in the flowgraph), data flow analysis, alias classification and pointer analysis, dead code elimination, copy propagation, partial redundancy elimination and strength reduction.

The remainder of the process depends on the target machine: for Itanium platforms, the code generator in Open64 can be directly used to generate object files. For a non-Itanium platform, the *whirl2c* or *whirl2f* translator will be invoked instead; in this case, code represented by Mid WHIRL is translated back to compilable, multithreaded C or Fortran code with OpenMP runtime calls. A native C or Fortran compiler must be invoked on the target platform to complete the translation by compiling the output from OpenUH into object files. The last step is the linking of object files with the portable OpenMP runtime library and final generation of executables for the target machine.

## 3 The Implementation of OpenMP

Based on our design and the initial status of Open64, we needed to focus our attention on developing or enhancing four major components in order to implement OpenMP: frontend extensions to parse OpenMP constructs and convert them into WHIRL IR with OpenMP pragmas, the internal translation of WHIRL IR with OpenMP directives into multithreaded code, a portable OpenMP runtime library supporting the execution of multithreaded code, and the IR-to-source translators, which needed work to enable them to generate compilable and portable source code.

To improve the stability of our frontends and to complement existing functionality, we integrated features from the Pathscale EKO 2.1 compiler. Its Fortran frontend contains many enhancements and the C/C++ frontend extends the more recent GCC 3.3 frontend with OpenMP parsing capability. The GCC parse tree is extended to represent OpenMP pragmas and is translated to WHIRL IR to enable later phases to handle it. The following subsections describe our OpenMP translation, runtime library and IR-to-source translators.

### 3.1 OpenMP Translation

An OpenMP implementation transforms code with OpenMP directives into corresponding multithreaded code with runtime library calls. A key component is the strategy for translating parallel regions. One popular method for doing so is outlining, which is used in most open source compilers, including Omni [4] and OdinMP/CCp [5]. Outlining denotes a strategy whereby an independent, separate function is generated by the compiler to encapsulate the work contained in a parallel region. In other words, a procedure is created that contains the code that will be executed by each participating thread at run time. This makes it easy to pass the appropriate work to the individual threads. In order to accomplish this, variables that are to be shared among worker threads have to be passed as arguments to the outlined function. Unfortunately, this introduces some overheads. Moreover, some compiler analyses and optimizations may be no longer applicable to the outlined function, either as a direct result of the separation into parent and outlined function or because the translation may introduce pointer references in place of direct references to shared variables.

The translation used in OpenUH is different from the standard outlining approach. In it, the compiler generates a microtask to encapsulate the code lexically contained within a parallel region, and the microtask is nested (we also refer to it as inlined, although this is not the standard meaning of the term) into the original function containing that parallel region. The advantage of this approach is that all local variables in the original function are visible to the threads executing the nested microtask and thus they are shared by default. Also, optimizing compilers can analyze and optimize both the original function and the microtask, thus providing a larger scope for intraprocedural optimizations than the outlining method. A similar approach named the Multi-Entry Threading (MET) technique [15] is used in Intel's OpenMP compiler.

Fig. 2 illustrates each of these strategies for a fragment of C code with a single parallel region, and shows in detail how the outlining method used in Omni differs from the inlining translation in OpenUH. In both cases, the compiler generates an extra function (the microtask $\_\_ompregion\_main1()$ or the outlined function $\_\_ompc\_func\_0()$) as part of the work of translating the parallel region enclosing $do\_sth(a, b, c)$. In each case, this function represents the work to be carried out by multiple threads. Each translation also adds a runtime library call ($\_\_ompc\_fork()$ or $\_ompc\_do\_parallel()$, respectively) into the main function, which takes the address of the compiler-generated function as an argument and executes it on several threads. The only extra work needed in the translation to the nested microtask is to create a thread-local variable to realize the private variable $c$ and to substitute this for $c$ in the call to the enclosed procedure, which now becomes $do\_sth(a, b, \_\_mylocal\_c)$. The translation that outlines the parallel region has more to take care of, since it must wrap the addresses of shared variables $a$ and $b$ in the main function and pass them to the runtime library call. Within the outlined procedure, they are referenced via pointers. This is visible in the call to the enclosed procedure, which in this version becomes $do\_sth(*\_pp\_a, *\_pp\_b, \_p\_c)$. The nested translation leads to shorter code and is more amenable to subsequent compiler optimizations.

**Original OpenMP Code**

```
int main(void)
{
  int a,b,c;

#pragma omp parallel private(c)
  do_sth(a,b,c);

  return 0;
}
```

**Inlined (Nested) Translation**

```
_INT32 main()
{
  int a,b,c;

/*inlined (nested) microtask */
void __ompregion_main1()
{
  _INT32 __mplocal_c;

/*shared variables are keep intact, only
substitute the access to private
variable*/
  do_sth(a, b, __mplocal_c);
}
 …
/*OpenMP runtime call */
 __ompc_fork(&__ompregion_main1);
  …
}
```

**Outlined Translation**

```
/*Outlined function with an extra argument
for passing addresses*/
static void __ompc_func_0(void
**__ompc_args){
    int *_pp_b, *_pp_a, _p_c;

/*dereference addresses to get shared
variables */
_pp_b=(int *)(*__ompc_args);
_pp_a=(int *)(*(__ompc_args+1));

/*substitute accesses for all variables*/
do_sth(*_pp_a,*_pp_b,_p_c);
}

int _ompc_main(void){
    int a,b,c;
    void *__ompc_argv[2];

/*wrap addresses of shared variables*/
*(__ompc_argv)=(void *)(&b);
*(__ompc_argv+1)=(void *)(&a);
…
/*OpenMP runtime call has to pass the
addresses of shared variables*/
_ompc_do_parallel(__ompc_func_0,
                __ompc_argv);
…
}
```

**Fig. 2.** OpenMP translation: outlined vs. inlined

Both the original Open64 and OpenUH precede the actual OpenMP translation with a preprocessing phase named OpenMP Prelowering, which facilitates later work by reducing the number of distinct OpenMP constructs that occur in the IR. It does so by translating some of them into others. (This smaller set of features is named MP in Open64 and they can also be generated by the Auto Parallelization module in LNO, enabling Open64 to support both automatic and manual parallelization in a common framework.) It also performs semantic checks. For example, a `barrier` is not allowed within a `critical` or `single` region. Some of the tasks performed are:

1. Converting `section` into `omp do`.
2. Converting unsupported *Fetch_And_Op* intrinsics such as *Fetch_And_Add* into `atomic`.
3. Inserting memory barriers around each parallel region to prevent impermissible code motion.
4. Lowering `atomic` using one of three possible ways: replacement by a `critical`, a *Compare_and_Swap* or *Fetch_And_Op*.

After prelowering, the remaining constructs are lowered. A few OpenMP directives can be handled by a one-to-one translation; they include `barrier`, `atomic` and `flush`. For example, we can replace `barrier` by a runtime library call named *_ompc_barrier*(). Most other OpenMP directives demand significant changes to the WHIRL tree, including rewriting the code segment and generating a new code segment to implement the multithreaded model.

The OpenMP standard makes the implementation of nested parallelism optional. The original Open64 chose to implement just one level of parallelism, which permits a straightforward multithreaded model. The implementation of nested parallelism in OpenUH is work in progress. When the master thread encounters a parallel region, it will check the current environment to find out whether it is possible to fork new threads. If so, the master thread will then fork the required number of worker threads to execute the compiler-generated microtask; if not, a serial version of the original parallel region will be executed by the master thread. Since only one level of parallelism is implemented, a parallel region within another parallel region is serialized in this manner.

Fig. 3 shows how a parallel region is translated. The compiler-generated nested microtask containing its work is named *__ompregion_main*1(), based on the code segment within the scope of the `parallel` directive in *main*(). It also rewrites the original code segment to implement its multithreaded model: this requires it to test via the corresponding OpenMP runtime routine whether it is already within a parallel region, in which case the code is executed sequentially. If not, and if threads are available, the parallel code version will be used. The parallel version contains a runtime library call named *__ompc_fork*() which takes the microtask as an argument. *__ompc_fork*() is the main routine from the OpenMP runtime library. It is responsible for manipulating worker threads and it assigns microtasks to them.

Fig. 4 shows how a code segment containing the worksharing construct `omp for`, which in this case is "orphaned" (i.e. is not within the lexical scope of the enclos-

| OMP PARALLEL | Code segment rewriting & microtask creation |
|---|---|
| #include <omp.h><br><br>int main(void)<br>{<br>#pragma omp parallel<br>printf("Hello,world.!\n");<br>} | int main(void)<br>{<br>/* inlined microtask generated from parallel region */<br>void __ompregion_main1( …)<br>{<br>printf(“Hello,world.!\n”);<br>return;<br>} /* __ompregion_main1 */<br>…..<br>/* Implement multithreaded model */<br>__ompv_in_parallel = __ompc_in_parallel();<br>__ompv_ok_to_fork = __ompc_can_fork();<br>if(((__ompv_in_parallel== 0) && (__ompv_ok_to_fork == 1)))<br>{<br>/* Parallel version: a runtime library call for creating<br>multiple threads and executing the microtask in parallel */<br>__ompc_fork(&__ompregion_main1,…);<br>} else<br>{ /* Sequential version */<br>printf(“Hello,world.!\n”);<br>return;<br>}<br>} |

**Fig. 3.** Code reconstruction to translate a parallel region

ing parallel construct), is rewritten. There is no need to create a new microtask for this orphaned `omp for` because it will be invoked from within the microtask created to realize its caller's parallel region. OpenMP parcels out sets of loop iterations to threads according to the schedule specified; in the static case reproduced here, a thread should determine its own execution set at run time. It does so by using its unique thread ID and the current schedule policy to compute its lower and upper loop bounds, along with the stride. A library call to retrieve the thread ID precedes this. The loop variable $i$ is private by default and so it has been replaced by the thread's private variable *__mplocal_i*. The implicit barrier at the end of the worksharing construct is also made explicit at the end of the microtask as required by the OpenMP specifications. Chen et al. [13] describe in more detail the classification of OpenMP directives and their corresponding transformation methods in the Open64 compiler.

Data environment handling is simplified by the adoption of nested microtasking instead of outlined functions to represent parallel regions. All global and local variables in the original function are visible to a nested microtask; the `shared` data attribute in OpenMP is thus available for free. Only `private` variables need to be translated. We have seen in the examples that this is achieved by creating temporary variables that are local to the thread and will be stored on the thread stacks at runtime. Variables in `firstprivate`, `lastprivate` and `reduction` lists are treated in a similar way, but require some additional work. First, a private variable is created. For `firstprivate`, the compiler adds a statement to initialize the local copy using the value of its global counterpart at the beginning of the code segment. For `lastprivate`, some code is added at the end to determine if the current iteration is the last one that would occur in the sequential code. If so, it transfers the value of the local copy to its global

| Orphaned OMP FOR | Rewriting the code segment |
|---|---|
| static void init(void)<br>{<br>int i;<br>#pragma omp for<br>for<br>(i=0;i<1000;i++)<br>{<br>a[i]= i*2;<br>}<br>} | void init()<br>{ …….<br>/* get current thread id */<br>__ompv_gtid_s = __ompc_get_thread_num();<br>……..<br>/* invoke static scheduler */<br>__ompc_static_init(__ompv_gtid_s, STATIC_EVEN,<br>&__ompv_do_lower,&__ompv_do_upper, &__ompv _do_stride, …);<br><br>/* execute loop body using assigned iteration space */<br>for(__mplocal_i = __ompv _do_lower; (__mplocal_i <= __ompv _do_upper); __mplocal_i = (__mplocal_i + 1))<br>{ a[__mplocal_i] = __mplocal_i*2;<br>}<br>/* Implicit BARRIER after work sharing constructs */<br>__ompc_barrier();<br>return;<br>} |

**Fig. 4.** Code reconstruction to translate an OMP FOR

counterpart. `reduction` variables are translated in two steps. In the first step, each thread performs its own local reduction operation. In the second step, the reduction operation is applied to combine the local reductions and the result is stored back in the global variable. To prevent a race condition, the compiler encloses the final reduction operation within a critical section. The handling of `threadprivate`, `copyin` and `copyprivate` variables is discussed below.

### 3.2 A Portable OpenMP Runtime Library

The role of the OpenMP runtime library is at least twofold. First, it must implement standard user level OpenMP runtime library routines such as *omp_set_lock*(), *omp_set_num_threads*() and *omp_get_wtime*(). Second, it should provide a layer of abstraction for the underlying thread manipulation (to perform tasks such as thread creation, suspension and wakeup) and deal with repetitive tasks (such as internal variable bookkeeping, calculation of chunks for each thread used in different scheduling options). The runtime library can free compiler writers from many tedious chores that arise in OpenMP translation and library writers can often conduct performance tuning without needing to delve into details of the compiler. All OpenMP runtime libraries are fairly similar in term of functionality, but the division of work between the compiler and runtime library is highly implementation-dependent. In other words, an OpenMP runtime library is tightly coupled with a particular OpenMP translation in a given compiler.

Our runtime library is based on the one shipped with the ORC-OpenMP compiler, which in turn borrowed some ideas from the Omni compiler's runtime library. Like most other open source ones, it relies on the Pthread API to manipulate underlying threads as well as to achieve portability. A major task of the runtime library is to create and manage threads in a team. When an OpenMP program starts to execute, the runtime library initialization is performed by the master thread when the first parallel region is encountered (this is indicated by

the API call $\_\_ompc\_fork()$). If N is the number of desired threads in the team, it will create N-1 worker threads and initialize internal variables (to record such things as the number of threads and the default scheduling method) related to the thread team. The worker threads will sleep until the master thread notifies them that a microtask is ready to be executed. The master then joins them to carry out the work of the microtask. The worker threads go back to sleep after finishing their microtask and will wait until they are notified of the next microtask. In this way, the worker threads are reused throughout the execution of the entire program and the overhead of thread creation is reduced to a minimum. This strategy is widely used in OpenMP implementations.

We enhanced the original ORC-OpenMP runtime library to support the compiler's implementation of the `threadprivate`, `copyin` and `copyprivate` clauses. For `threadprivate` variables, the runtime library will dynamically allocate private copies on the heap storage for each thread and store their start addresses in an array indexed by thread IDs. Thus each thread can easily access its own copy of the data and the values may persistent across different parallel regions. `copyin` is implemented via binary copy from the global value of a `threadprivate` variable to the current thread's private copy in the heap storage. To implement `copyprivate`, a new internal variable is introduced to store the address of the `copyprivate` variable from the `single` thread and all other threads will copy the value by dereferencing it. Some extra attention is needed to ensure the correct semantics: a barrier is used to ensure all other threads do not copy the value before the `single` thread has set the address. Another barrier is used to ensure the `single` thread will not proceed until all other threads finish the copying.

Other enhancements to the runtime library include changing some interfaces to accommodate new translations, optimizing the division of the work between the compiler and runtime library, modification to improve its portability, and performance tuning. The resulting OpenMP runtime library is now one of the most complete open source implementations.

### 3.3 The IR-to-Source Translators

We considered it essential that the compiler be able to generate code for a variety of platforms. We initially attempted to translate Mid WHIRL to the GNU RTL, but abandoned this approach after it appeared to be too complex. Instead, we adopted a source-to-source approach and enhanced the IR-to-source translators that came with the original Open64 (*whirl2c* and *whirl2f*) to output compilable, portable C and Fortran source code after translating OpenMP. As previously described, a native C or Fortran compiler can then generate the object files and link them with the portable OpenMP runtime library on non-Itanium platforms.

To achieve this, the original *whirl2c* and *whirl2f* had to be extended to translate Mid level WHIRL to compilable code after the WOPT phase. This approach preserves valuable analysis and optimizations as far as possible. This created many challenges, as the *whirl2c/whirl2f* tools were only designed to help compiler developers look at the High level WHIRL corresponding to a submitted program in a human-readable way. We required them to emit compilable and

portable source code. For example, the compiler-generated nested function to realize an OpenMP parallel region was output by *whirl2c/whirl2f* as a top-level function, since the compiler works on program units one at a time and does not treat these in a special way; this will not compile correctly, since in particular, the shared variables will be undefined. To handle this particular problem, a new phase was added to *whirl2c/whirl2f* to restore the nested semantics for micro-tasks using the nested function supported by GCC and `CONTAINS` from Fortran 90. Another problem is that though most compiler transformations before the CG (code generation) phase are machine-independent, some of them still take platform-specific parameters or make hardware assumptions, such as expecting dedicated registers to pass function parameters, the transformation for 64-bit ISA, and register variable identification in WOPT. To deal with this, a new compiler option `-portable` has been introduced to let the compiler perform only portable phases or to apply translations in a portable way (for instance, the OpenMP `atomic` construct will be transformed using the `critical` construct rather than using machine-specific instructions). Some other problems we faced included missing headers, an incorrect translation for multidimensional arrays, pointers and structures, and incompatible data type sizes for 32-bit and 64-bit platforms. We used the enhanced *whirl2c* tool from the Berkeley UPC compiler to help resolve some of these problems.

## 4 Evaluation of The Compiler

We have chosen a set of benchmarks and platforms to help us evaluate the compiler for correctness, performance and portability. The major platform used for testing is COBALT, an SGI Altix system at NCSA. Cobalt is a ccNUMA platform with a total of 1024 1.6GHz Itanium 2 processors with 1024 or 2048 GB memory. Two other platforms were also used: an IA32 system running Redhat 9 Linux with dual Xeon-HT 2.4 GHZ CPUs and 1.0GB memory, and a SunFire 880 node from the University of Houston's Sun Galaxy Cluster, running Solaris 9 with four 750MHz UltraSPARC-III processors and 8 GB memory. The source-to-source translation method is used when the platform is not Itanium-based.

The correctness of the OpenMP implementation in the OpenUH compiler was our foremost consideration. To determine this, we used the public OpenMP validation suite [16] to test the compiler's support for OpenMP. All OpenMP 1.0 and 2.0 directives and most of their legal combinations are included in the tests. Results on the three systems showed our compiler passed almost all tests and had verified results. But we did notice some unstable results from the test for `single_copyprivate` and this is under investigation.

The next concern is to measure the overheads of the OpenUH compiler translation of OpenMP constructs. The EPCC microbenchmark [17] has been used for this purpose. Fig. 5 and Fig. 6 show the parallel overheads and scheduling overheads, respectively, of our compiler on 1 to 8 threads on COBALT. All constructs have acceptable overheads except for `reduction`, which uses a `critical`

section to protect the reduction operation on local values for each thread to obtain portability.

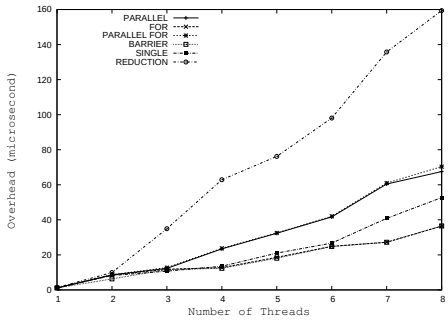


**Fig. 5.** Parallel overheads of OpenUH on COBALT

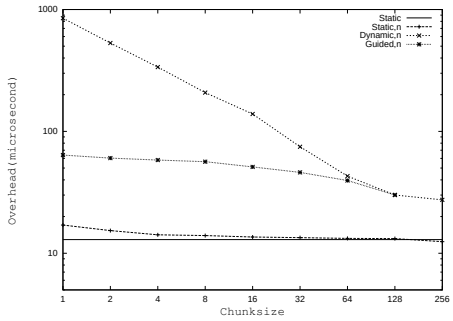


**Fig. 6.** Scheduling overheads of OpenUH on COBALT

We used the popular NAS parallel benchmark (NPB) [18] to compare the performance of OpenUH with two other OpenMP compilers: the commercial Intel 8.0 compiler and the open source Omni 1.6 compiler. A subset of the latest NPB 3.2 was compiled using the Class A data set by each of the three compilers. The compiler option - `O2` was used and the executables were run on 4 threads on COBALT. Fig. 7 shows the normalized Mop/s ratio for seven benchmarks. The results of LU and LU-HP from Omni were not verified but we include the performance data for a more complete comparison. OpenUH outperformed Omni except for the EP benchmark. Despite its reliance on a runtime system designed for portability rather than highest performance on a given platform, OpenUH even achieved better performance than the Intel compiler in several instances, as demonstrated by the FT and LU-HP benchmarks. The result of this test confirms that the OpenUH compiler can be used as a serious research OpenMP compiler on Itanium platforms.

The evaluation of portability and effectiveness of preserved optimizations using the source-to-source approach has been conducted on all three test machines. The native GCC compiler on each machine is used as a backend compiler to compile the multithreaded code and link the object files with the portable OpenMP runtime library. We compiled NPB 2.3 OpenMP/C in three ways: using no optimization in both OpenUH compiler and the backend GCC compiler, using `O3` for GCC only, and using `O3` for both OpenUH and GCC compilers. All versions were executed with dataset A on four threads. Fig. 8 shows the speedup of the CG benchmark using different optimization levels on the three platforms. Other benchmarks have similar speedup but are not shown here due to space limits. The version with optimizations from both OpenUH and GCC achieves thirty (Itanium and UltraSparc) to seventy percent (Xeon) extra speedup over the version with only GCC optimizations, which means the optimizations from

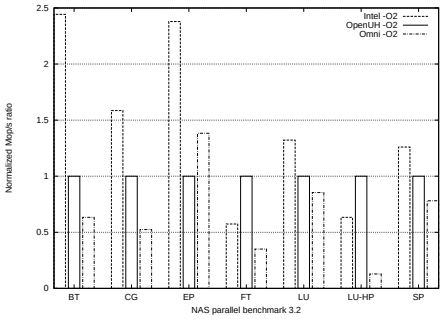


**Fig. 7.** Performance comparison of several compilers using NPB 3.2

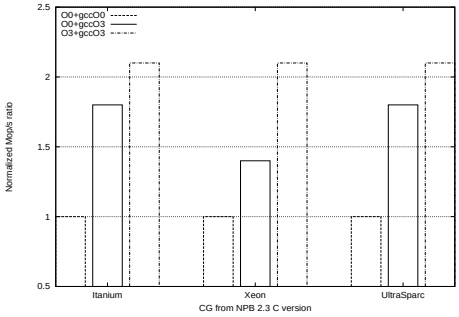


**Fig. 8.** Speedup of CG using whirl2c with Optimizations

OpenUH are well preserved under the source-to-source approach and have a significant effect on the final performance on multiple platforms.

## 5 Related Work

Almost all major commercial compilers support OpenMP today. Most target specific platforms for competitive performance. They include Sun Studio, Intel compiler, Pathscale EKO compiler suite and Microsoft Visual Studio 2005 beta. Most are of limited usage for public research. Pathscale's EKO compiler suite is open source because it is derived from the GPL'ed SGI Pro64. It is a good reference OpenMP implementation. However, its OpenMP runtime library is proprietary and targets the AMD X8664 platform.

Omni [4] is a popular source-to-source translator from Tsukuba University supporting C/Fortran 77 with a portable OpenMP runtime library based on POSIX and Solaris threads. But it has little program analysis and optimization ability and does not yet support OpenMP 2.0. OdinMP/CCp [5] is another source-to-source translator with only C language support. NanosCompiler [19] tries to combine automatic parallelization with manual parallelism annotations using OpenMP. It also implements a variety of extensions to OpenMP. However, it is not a fully functional OpenMP compiler and the source is not released. The ORC-OpenMP compiler [13] can be viewed as a sibling of the OpenUH compiler in view of the common source base. But its C/C++ frontend, based on GCC 2.96, is not yet stable and some important OpenMP constructs (e.g. `threadprivate`) are not implemented. It targets the Itanium. PCOMP [6] contains an OpenMP parallelizer and a translator to generate portable multithreaded code to be linked with a runtime library. Unfortunately, only Fortran 77 is supported. GOMP [20] is an ongoing project to provide OpenMP support in the GCC compiler. The Berkeley UPC compiler effort [11] uses a similar idea to ours. Our compiler has integrated and enhanced many desirable features from Pathscale, ORC-OpenMP and the Berkeley UPC compiler.

## 6 Conclusions and Future Work

In this paper, we have presented our effort to create an optimizing, portable OpenMP compiler based on the Open64 compiler infrastructure and its branches. The result is a complete implementation of OpenMP 2.5 on Itanium platforms. It also targets other platforms by providing a source-to-source translation path with a portable OpenMP runtime library. Extensive tests have been applied to evaluate our compiler, including the OpenMP validation suite, the EPCC microbenchmarks and the NAS parallel benchmarks. Its features offer numerous opportunities to explore further enhancements to OpenMP and to study its performance on existing and new architectures. Our experience also demonstrates that the open source Open64 compiler infrastructure is a very good choice for compiler research, given the modularized infrastructure and code contributions from different organizations.

In the future, we will focus on performance tuning both the OpenMP translation and the runtime library. We intend to support nested parallelism. We are using OpenUH to explore language features that permit subsets of a team of threads to execute code within a parallel region, which would enable several subteams to execute concurrently [21]. Enhancing existing compiler optimizations to improve OpenMP performance on new chip multithreading architectures is also a focus of our investigation [2]. We are exploring the creation of cost models within the compiler to help detect resource conflicts among threads and obtain better thread scheduling. Meanwhile, we are considering an adaptive scheduler to improve the scalability of OpenMP on large scale NUMA systems.

## Acknowledgments

This work is funded by National Science Foundation under contract CCF-0444468 and Department of Energy under contract DE-FC03-01ER25502. We thank all contributors to the Open64 and ORC compilers. The Sun Microsystems Center of Excellence in the Geosciences at the University of Houston and the National Center for Supercomputing Applications (NCSA) at University of Illinois at Urbana-Champaign provided access to high-performance computing resources.

## References

1. http://www.openmp.org: OpenMP: Simple, portable, scalable SMP programming (2005)
2. Liao, C., Liu, Z., Huang, L., Chapman, B.: Evaluating OpenMP on chip multithreading platforms. In: First international workshop on OpenMP, Eugene, Oregon USA (2005)
3. Huang, L., Chapman, B., Kendall, R.: OpenMP on distributed memory via Global Arrays. In: Parallel Computing 2003 (PARCO 2003), DRESDEN, Germany (2003)
4. Sato, M., Satoh, S., Kusano, K., Tanaka, Y.: Design of OpenMP compiler for an SMP cluster. In: the 1st European Workshop on OpenMP(EWOMP'99). (1999) 32–39

5. Brunschen, C., Brorsson, M.: OdinMP/CCp - a portable implementation of OpenMP for C. Concurrency - Practice and Experience **12** (2000) 1193–1203
6. Min, S.J., Kim, S.W., Voss, M., Lee, S.I., Eigenmann, R.: Portable compilers for OpenMP. In: WOMPAT '01: Proceedings of the International Workshop on OpenMP Applications and Tools, London, UK, Springer-Verlag (2001) 11–19
7. http://www.cs.uh.edu/∼openuh: The OpenUH compiler project (2005)
8. http://open64.sourceforge.net/: The Open64 compiler (2005)
9. http://gcc.gnu.org/: the gnu compiler collection (2005)
10. http://ipf orc.sourceforge.net/: Open research compiler for itanium processor family (2005)
11. Chen, W.Y.: Building a source-to-source UPC-to-C translator. Master's thesis, University of California at Berkeley (2005)
12. Carlson, W.W., Draper, J.M., Culler, D.E., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and language specification. Technical report, Center for Computing Sciences (1999)
13. Chen, Y., Li, J., Wang, S., Wang, D.: ORC-OpenMP: An OpenMP compiler based on ORC. In: International Conference on Computational Science. (2004) 414–423
14. http://www.pathscale.com/ekopath.html: Pathscale compiler suite (2005)
15. Tian, X., Bik, A., Girkar, M., Grey, P., Saito, H., Su, E.: Intel OpenMP C++/Fortran compiler for hyper-threading technology: Implementation and performance. Intel Technology Journal **6** (2002) 36–46
16. Müller, M.S., Niethammer, C., Chapman, B., Wen, Y., Liu, Z.: Validating OpenMP 2.5 for Fortran and C/C++. In: Sixth European Workshop on OpenMP, KTH Royal Institute of Technology, Stockholm, Sweden (2004)
17. Bull, J.M., O'Neill, D.: A microbenchmark suite for OpenMP 2.0. In: Proceedings of the Third European Workshop on OpenMP (EWOMP'01), Barcelona, Spain (2001)
18. Jin, H., Frumkin, M., Yan, J.: The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Ames Research Center (1999)
19. Ayguadé, E., Gonzàlez, M., Martorell, X., Oliver, J., Labarta, J., Navarro, N.: NANOSCompiler: A research platform for OpenMP extensions. In: the First EuropeanWorkshop on OpenMP, Lund, Sweden (1999) 27–31
20. http://gcc.gnu.org/projects/gomp/: GOMP - an OpenMP implementation for GCC (2005)
21. Chapman, B.M., Huang, L., Jost, G., Jin, H., de Supinski, B.R.: Support for flexibility and user control of worksharing in OpenMP. Technical report, National Aeronautics and Space Administration (2005) NAS Technical Report NAS-05-015, October 2005.