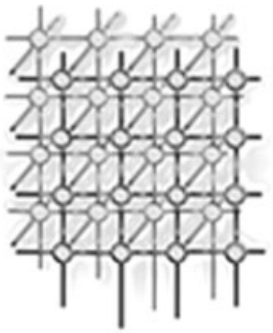
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2007; **19**:2317–2332
Published online 3 April 2007 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1174

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, TX 77004, U.S.A.*
[2]*Department of Computer Science and Technology, Tsinghua University, Beijing 100084, People's Republic of China*

SUMMARY

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, for example into its use on new target architectures, such as SMPs with chip multi-threading, 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. Copyright © 2007 John Wiley & Sons, Ltd.

Received 28 February 2006; Revised 2 December 2006; Accepted 23 December 2006

KEY WORDS: hybrid OpenMP compiler; OpenMP translation; OpenMP runtime library

1. INTRODUCTION

OpenMP [1], a set of compiler directives, runtime library routines and environment variables, 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

*Correspondence to: Barbara Chapman, Computer Science Department, University of Houston, 4800 Calhoun Road, Houston, TX 77004, U.S.A.
†E-mail: chapman@cs.uh.edu

Contract/grant sponsor: Department of Energy; contract/grant number: DE-FC03-01ER25502
Contract/grant sponsor: National Science Foundation; contract/grant number: CCF-0444468

Copyright © 2007 John Wiley & Sons, Ltd.

WILEY InterScience® DISCOVER SOMETHING GREAT

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 multi-threading 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 proprietary products such as the Intel compilers, Sun Studio compilers 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], Mercurium [6], and PCOMP [7]) are available. However, none of them translate all of the source languages that OpenMP supports, and most of them [4–6] are source-to-source translators that have little analysis and optimization ability. 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 from [8]) 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 and 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 suggestions for future work.

2. THE DESIGN OF OPENUH

OpenMP is a fork-join parallel programming model with bindings for C/C++ and Fortran 77/90 to provide additional shared memory parallel semantics. The OpenMP extensions consist primarily of compiler directives (structured comments that are understood by an OpenMP compiler) for the creation of parallel programs; these are augmented by user-level runtime routines and environment variables. The major OpenMP directives enable the program to create a team of threads to execute a specified region of code in parallel (`omp parallel`), to share out work in a loop or in a set of code segments (`omp do` (or `omp for`) and `sections`) and to provide data environment management (`private` and `shared`) and thread synchronization (`barrier`, `critical` and `atomic`). User-level runtime routines allow users to detect the parallel context (*omp_in_parallel*()), check and adjust the number of executing threads (*omp_get_num_threads*() and *omp_set_num_threads*()) and use locks (*omp_set_lock*()). Environment variables may also be used to adjust the runtime behavior of OpenMP applications, particularly by setting defaults for the current run. For example, it is possible to set the default number of threads to execute a parallel region (`OMP_NUM_THREADS`) and the default scheduling policy (`OMP_SCHEDULE`) etc.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

It is a very reasonable proposition to create an OpenMP compiler. Indeed, a large fraction of the work of implementing this API is within the runtime library support. Research compilers are typically stand-alone source-to-source compilers that translate source code with OpenMP constructs into explicitly multi-threaded code making calls to a portable OpenMP runtime library, as exemplified in [4,5]. Then, a back end compiler is needed to continue the compilation and generate executables. The source-to-source translation approach has the benefit of portability and thus permits the resulting compiler to be widely used; it is by far the most convenient approach for academia. However, the unfortunate lack of interactions between the two distinct compilers, who naturally do not share any program analysis, may potentially have a negative impact on achievable performance. In contrast, most commercial compilers enable OpenMP using a module integrated within their standard optimizing infrastructures. In this case, the OpenMP translation has access to program analysis information and can work tightly with all other phases. The integrated approach often leads to better performance, but does not provide portability and is rather difficult to achieve in an academic setting, where the back end infrastructure is seldom available. An interesting problem for OpenMP compilation is the placement of the OpenMP module in the overall translation process, and interactions between it and other analysis and optimization phases. The existing phases designed for sequential code have to be re-examined under the parallel context to ensure correctness and performance. One of the major challenges is to ensure that the use of OpenMP does not impair traditional sequential optimizations, such as loop nest transformations, to better exploit cache.

For our compiler implementation we decided to realize a hybrid approach, where portability is achieved via a source-to-source translation but where we also have a complete and integrated OpenMP-aware compiler in order to evaluate our results in a setting that is typical of industrial compilers. Within the full compiler we are better able to experiment to determine the impact of moving the relative position of the OpenMP translation, and can experiment with a variety of strategies for handling loop nests. Given the 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 [9] compiler suite, which we judged to be more suitable than, in particular, the GNU Compiler Collection [10] in its current state of development. Open64 was open sourced by Silicon Graphics Inc. and is now mostly maintained by Intel under the name Open Research Compiler (ORC) [11]. It targets Itanium platforms. It 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 front ends, 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.

Figure 1 depicts an overview of the design of OpenUH based on Open64. It consists of the front ends, 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 front end and translated into WHIRL IR with OpenMP pragmas. The next phase, the IPA, is enabled if desired to carry out interprocedural alias analysis,

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

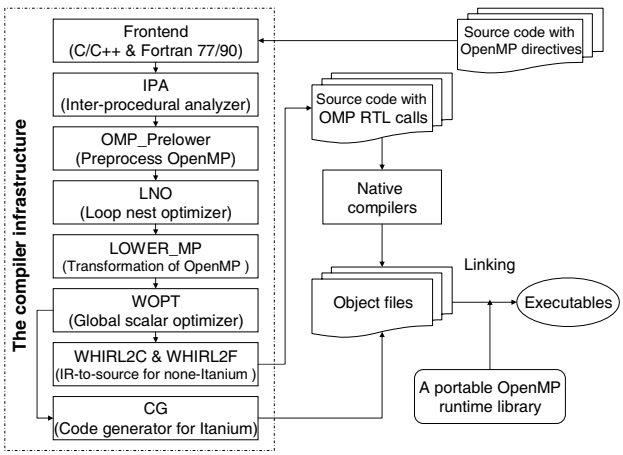


Figure 1. OpenUH: an optimizing and portable OpenMP compiler based on Open64.

array section analysis, inlining, dead function and variable elimination, interprocedural constant propagation and more. After that, the 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 transformation of OpenMP, which lowers WHIRL with OpenMP pragmas into WHIRL representing multi-threaded code with OpenMP runtime library calls, is performed in two steps: OMP_Prelower and LOWER_MP. The first phase preprocesses OpenMP pragmas while the latter does the major transformation. The LOWER_MP phase may occur before or after LNO although the default choice is after LNO as shown in Figure 1. The global scalar optimizer (WOPT) is subsequently invoked. It transforms WHIRL into an SSA (static single assignment) form for more efficient analyses 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 analyses (computing dominance, detecting loops in the flow graph), 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, compilable, multi-threaded C or Fortran code with OpenMP runtime calls is generated from mid WHIRL code instead. A native C or Fortran compiler must be invoked on the target platform to complete the translation.

The resulting compiler works in two modes: as a fully fledged native compiler on Itanium platforms for best performance and as a portable source-to-source translator with considerable optimizations for the others. This hybrid design permits us to achieve portability, and yet also to perform experiments that use the complete compiler where a native back end is available. We are able to exploit most of the compiler's optimizations right up to the code generator.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

3. THE IMPLEMENTATION

Despite the good match between Open64 infrastructure and our design goals, we faced many practical challenges during the implementation of OpenUH: there was no support for OpenMP directives in the C/C++ front end, which was taken from GCC 2.96. The Fortran parts of the compiler provided partial support for OpenMP 1.0. Meanwhile, no corresponding OpenMP runtime library was released with Open64. The *whirl2c*/*whirl2f* translator was only designed for debugging high-level WHIRL, not for generating portable and compilable source code from mid-level WHIRL. Finally, some compiler transformations such as register variable recognition and machine-specific intrinsics for the ATOMIC construct were Itanium platform-specific and needed to be reinvestigated for portability.

Two of the authors of this paper tackled some of the open problems by extending Open64's C front end to parse OpenMP constructs and by providing a tentative runtime library. This work was released in the context of the ORC-OpenMP [12] compiler from Tsinghua University. At the University of Houston, the remaining authors of this paper independently began working on the Open64.UH compiler effort, which focused more 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. Several other Open64 source code branches helped to facilitate our implementation. Recently, a commercial product based on Open64 and targeting the AMD x8664, the Pathscale EKO compiler suite [13], was released with support for OpenMP 2.0. The Berkeley UPC compiler [14] extends Open64 to implement UPC [15] by enhancing the *whirl2c* translator.

Based on our design and the initial status of Open64, we focused our attention on developing or enhancing four major components of Open64: front end extensions to parse OpenMP constructs and convert them into WHIRL IR with OpenMP pragmas, the internal translation of WHIRL IR with OpenMP directives into multi-threaded code, a portable OpenMP runtime library supporting the execution of multi-threaded code and the IR-to-source translators.

To improve the stability of our front ends and to complement existing functionality, we integrated features from the Pathscale EKO 2.1 compiler. The following subsections describe our work in OpenMP translation, the runtime library and the IR-to-source translators.

3.1. OpenMP translation

An OpenMP implementation transforms code with OpenMP directives into corresponding multi-threaded 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 runtime. This makes it easy to pass the appropriate work to the individual threads. In order to accomplish this, local 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 functions 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 this translation, the compiler generates a microtask to encapsulate the code lexically contained within a parallel region,

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

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);
…
}
```

Figure 2. OpenMP translation: outlined versus inlined.

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, and thus provide a larger scope for intraprocedural optimizations than the outlining method. A similar approach named the multi-entry threading (MET) technique [16] is used in Intel's OpenMP compiler.

Figure 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*

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

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.

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 (such as converting `section` into `omp do`). It also performs semantic checks. For example, a `barrier` is not allowed within a `critical` or `single` region. 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 multi-threaded 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 multi-threaded 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.

Figure 3 shows how a parallel region is translated. The compiler-generated nested microtask containing its work is named *__ompregion_main1*(), based on the code segment within the scope of the `parallel` directive in *main*(). It also rewrites the original code segment to implement its multi-threaded 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. The *__ompc_fork*() library is the main routine from the OpenMP runtime library. It is responsible for manipulating worker threads and it assigns microtasks to them.

Figure 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 enclosing 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 runtime. 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.* [12] describe in more detail the classification of OpenMP directives and their corresponding transformation methods in the Open64 compiler.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

OMP PARALLEL	Code segment rewriting & microtask creation

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

Figure 3. Code reconstruction to translate a parallel region.

Orphaned OMP FOR	Rewriting the code segment

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

Figure 4. Code reconstruction to translate an OMP FOR.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

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 during the compiler transformation. The `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 counterpart. The `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, synchronization, 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 terms of the basic 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 that shipped with the ORC-OpenMP compiler, which in turn borrowed some ideas from the Omni compiler's runtime library. As with most other open source libraries, it relies on the Pthreads API to manipulate underlying threads as well as to achieve portability. A major task of the runtime library is to create threads and assign microtasks to them in a team. When an OpenMP program starts to execute, the runtime library initialization is performed once 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.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

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 persist across different parallel regions. The `copyin` clause 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.

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 register transfer language, 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*) as mentioned.

To achieve portability and preserve valuable analyses and optimizations as far as possible, the original *whirl2c* and *whirl2f* had to be extended to translate mid-level WHIRL to compilable code after the WOPT phase. 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 compiled program in a human-readable way. We required them to be more powerful 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 by a back end compiler 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 microtasks using the nested function supported by GCC and `CONTAINS` from Fortran 90. Another problem is that though most compiler transformations before the CG 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 multi-dimensional arrays, pointers and structures and incompatible data type sizes for 32-bit and 64-bit platforms. We merged the enhanced *whirl2c* tool from the Berkeley UPC compiler [14] into OpenUH to help resolve some of these problems.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

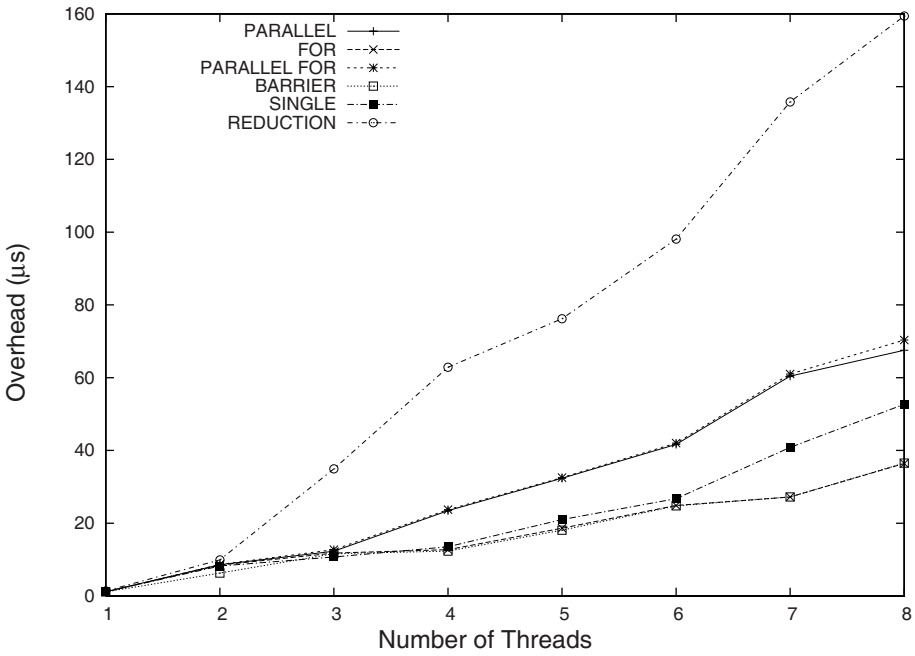


Figure 5. Parallel overheads of OpenUH.

4. EXPERIMENTAL EVALUATION

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 1.6 GHz Itanium 2 processors with 1024 or 2048 GB memory. Two other platforms were also used: an IA-32 system running Redhat 9 Linux with dual Xeon-HT 2.4 GHz CPUs and 1.0 GB memory; and a SunFire 880 node from the University of Houston's Sun Galaxy Cluster, running Solaris 9 with four 750 MHz 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 a public OpenMP validation suite [17] 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. However, 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 [18] has been used for this purpose. Figures 5 and 6 show the

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

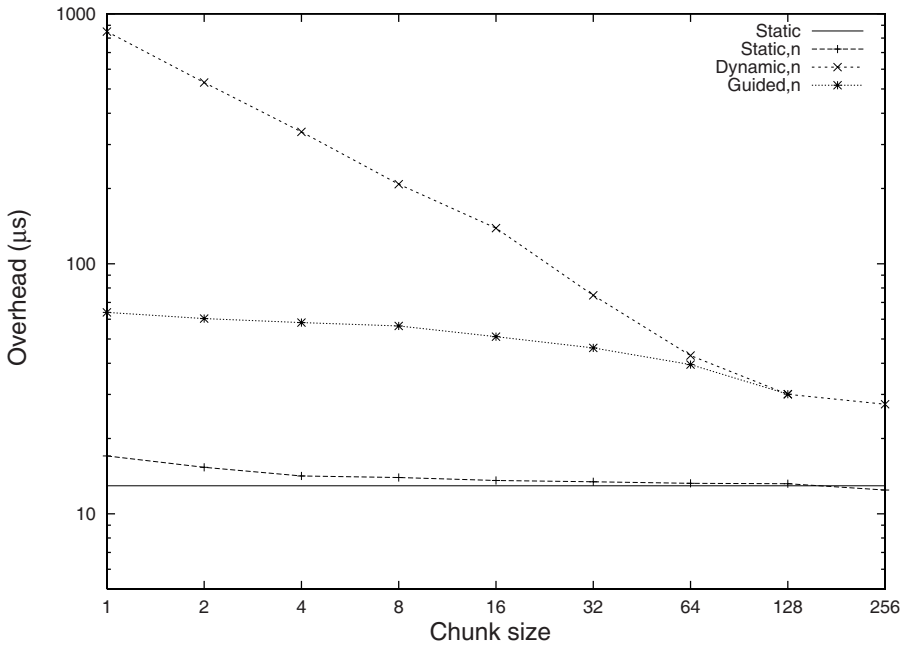


Figure 6. Scheduling overheads of OpenUH.

parallel overheads and scheduling overheads, respectively, of our compiler on one to eight threads on COBALT. Compared to the overhead diagrams shown in [18], 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.

We used the popular NAS parallel benchmark (NPB) [19] 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 one to 16 threads on COBALT. Figure 7 shows the normalized million operations per second (MOP s^{-1}) ratio for seven benchmarks using eight threads, which was representative. 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 the 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 the effectiveness of preserved optimizations using the source-to-source approach has been conducted on all three test machines. The native GCC compiler on each

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

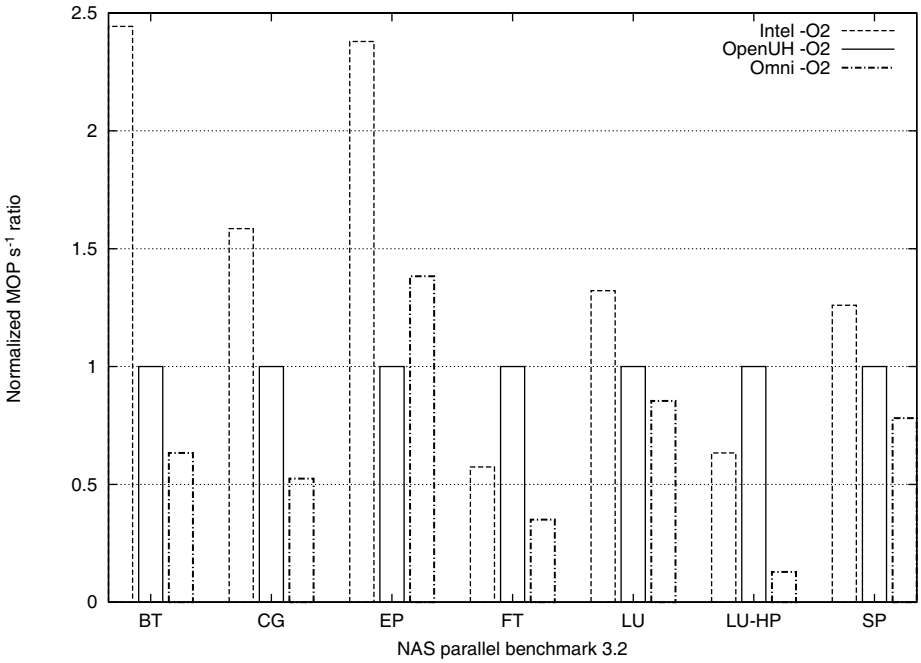


Figure 7. Performance comparison of several compilers.

machine is used as a back end compiler to compile the multi-threaded 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 either the OpenUH compiler or the back end GCC compiler; using `O3` for GCC only; and using `O3` for both OpenUH and GCC compilers. Also, the Omni compiler and two native commercial compilers (the Intel compiler 8.0 on Itanium and Xeon, and Sun Studio 10 on UltraSparc) were used to compare the performance of our source-to-source translation. The compilation options were all set to `O3` for those reference compilers whenever possible. All versions were executed with dataset A on four threads. Figure 8 shows the speedup of the CG benchmark using different optimization levels of OpenUH on the three platforms, along with the speedup using the reference compilers. Other benchmarks have similar speedup but are not shown here owing to space limits. The version with optimizations from both OpenUH and GCC achieves from 30% (for Itanium and UltraSPARC) to 70% (for Xeon) extra speedup over the version with only GCC optimizations, which means the optimizations from OpenUH are well preserved under the source-to-source approach and have a significant effect on the final performance on multiple platforms. OpenUH's source-to-source translation outperforms the Omni compiler on all three platforms, although this is not obvious on the Itanium machine. We believe OpenUH can do even better after some performance tuning in the OpenMP runtime library. It is also observed that the performance of OpenUH's native compilation model with `O3` is, on average, 30% higher than the source-to-source compilation strategy with `O3` on

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

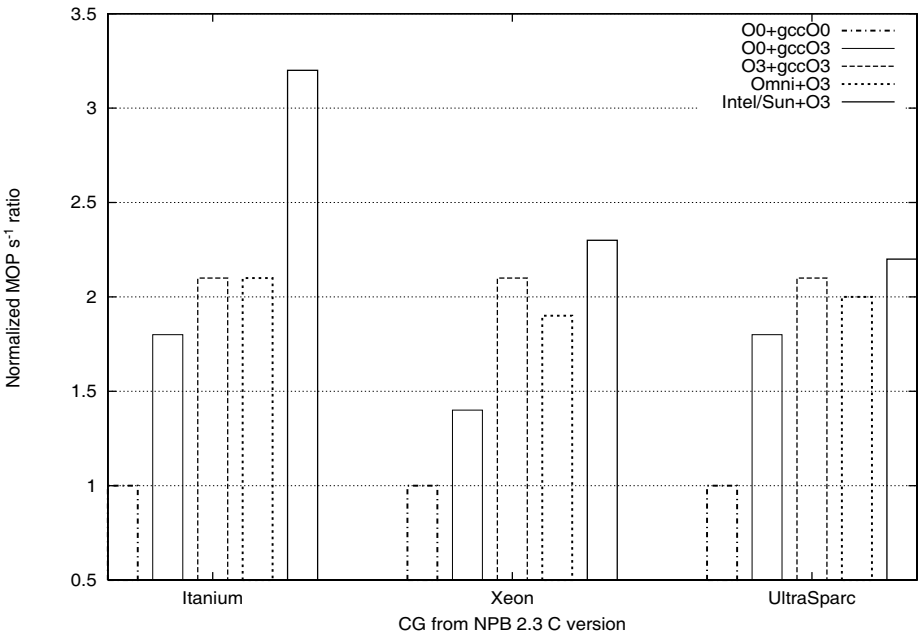


Figure 8. Using whirl2c with optimizations.

Itanium according to NPB 2.3 benchmarks, which is understandable given that the quality of Open64's code generator is higher than that of GCC.

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. 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, it targets the AMD X8664 platform and its OpenMP runtime library is proprietary.

Several OpenMP research compilers also exist. 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. However, 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 [20] is a source-to-source OpenMP compiler for Fortran 77. It also implements a variety of extensions to OpenMP including multi-level parallelization. However, it is not

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

a fully functional OpenMP compiler and the source is not released. The ORC-OpenMP compiler [12] can be viewed as a sibling of the OpenUH compiler in view of the common source base. However, its C/C++ front end, based on GCC 2.96, is not yet stable and some important OpenMP constructs (e.g. `threadprivate`) are not implemented. It targets the Itanium only. PCOMP [7] contains an OpenMP parallelizer and a translator to generate portable multi-threaded code to be linked with a runtime library. Unfortunately, only Fortran 77 is supported. NANOS Mercurium [6], another source-to-source translator for C and Fortran, explores template-based OpenMP translation. GOMP [21] is an ongoing project to provide OpenMP support in the GCC compiler.

In addition to our efforts, several other compilers [14,22,23] have used Open64 as the basic infrastructure for research goals in language and compiler research. The Kylin compiler [22] is a C compiler retargeting Open64's back end to Intel's Xscale architecture. Rice University [23] implements Co-Array Fortran based on Open64. The Berkeley UPC compiler effort [14] uses a similar approach to ours to support UPC.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented our efforts 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 new and existing 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 currently 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 [24]. Enhancing existing compiler optimizations to improve OpenMP performance on new chip multi-threading architectures is also a focus of our investigation [2]. We are also exploring the creation of cost models within the compiler to help detect resource conflicts among threads and obtain better thread scheduling. In addition, we are also considering an adaptive scheduler to improve the scalability of OpenMP on large-scale NUMA systems.

ACKNOWLEDGEMENTS

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 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.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

REFERENCES

1. OpenMP. Simple, portable, scalable SMP programming. http://www.openmp.org [June 2006].
2. Liao C, Liu Z, Huang L, Chapman B. Evaluating OpenMP on chip multithreading platforms. *Proceedings of the 1st International Workshop on OpenMP*, Eugene, OR, June 2005. Available at: http://www.nic.uoregon.edu/iwomp2005/Papers/f26.pdf [January 2007].
3. Huang L, Chapman B, Kendall R. OpenMP on distributed memory via global arrays. *Proceedings of the 2003 Conference on Parallel Computing (PARCO 2003)*, Dresden, Germany, 2003. Elsevier: Amsterdam, 2003.
4. Sato M, Satoh S, Kusano K, Tanaka Y. Design of OpenMP compiler for an SMP cluster. *Proceedings of the 1st European Workshop on OpenMP (EWOMP'99)*, September 1999; 32–39. Available at: http://www.it.lth.se/ewomp99/papers/sato.pdf [January 2007].
5. Brunschen C, Brorsson M. OdinMP/CCp—a portable implementation of OpenMP for C. *Concurrency—Practice and Experience* 2000; **12**(12):1193–1203.
6. Balart J, Duran A, Gonzalez M, Martorell X, Ayguade E, Labarta J. Nanos mercurium: A research compiler for OpenMP. *Proceedings of the 6th European Workshop on OpenMP (EWOMP'04)*, Stockholm, Sweden, October 2004. Available at: http://www.imit.kth.se/ewomp2004/proceedings.pdf [January 2007].
7. Min SJ, Kim SW, Voss M, Lee SI, Eigenmann R. Portable compilers for OpenMP. *Proceedings of the International Workshop on OpenMP Applications and Tools (WOMPAT'01)*, London, UK. Springer: Berlin, 2001; 11–19.
8. The OpenUH Compiler Project. http://www.cs.uh.edu/∼openuh [December 2006].
9. The Open64 Compiler. http://open64.sourceforge.net [November 2006].
10. The GNU Compiler Collection. http://gcc.gnu.org [November 2006].
11. Open Research Compiler for Itanium Processor Family. http://ipf-orc.sourceforge.net [September 2005].
12. Chen Y, Li J, Wang S, Wang D. ORC-OpenMP: An OpenMP compiler based on ORC. *Proceedings of the International Conference on Computational Science*, 2004; 414–423.
13. Pathscale EKOPATH compiler suite for AMD64 and EM64T. http://www.pathscale.com/ekopath.html [May 2006].
14. Chen W-Y. Building a source-to-source UPC-to-C translator. *Master's Thesis*, University of California at Berkeley, 2005.
15. Carlson WW, Draper JM, Culler DE, Yelick K, Brooks E, Warren K. Introduction to UPC and language specification. *Technical Report CCS-TR-99-157*, Center for Computing Sciences, May 1999.
16. 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* 2002; **6**:36–46.
17. Müller MS, Niethammer C, Chapman B, Wen Y, Liu Z. Validating OpenMP 2.5 for Fortran and C/C++. *Proceedings of the 6th European Workshop on OpenMP*, KTH Royal Institute of Technology, Stockholm, Sweden, October 2004. Available at: http://www.imit.kth.se/ewomp2004/proceedings.pdf [January 2007].
18. Bull JM, O'Neill D. A microbenchmark suite for OpenMP 2.0. *Proceedings of the 3rd European Workshop on OpenMP (EWOMP'01)*, Barcelona, Spain, September 2001.
19. 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. Available at: http://www.compunity.org/events/pastevents/ewomp2001/bull.pdf [January 2007].
20. Ayguadé E, Gonzàlez M, Martorell X, Oliver J, Labarta J, Navarro N. NANOSCompiler: A research platform for OpenMP extensions. *Proceedings of the 1st European Workshop on OpenMP*, Lund, Sweden, October 1999; 27–31. Available at: http://www.it.lth.se/ewomp99/papers/ayguade.pdf [January 2007].
21. GOMP. An OpenMP implementation for GCC. http://gcc.gnu.org/projects/gomp [November 2005].
22. Kylin C Compiler. http://www.capsl.udel.edu/kylin/ [July 2006].
23. Open64 Project at Rice University. http://hipersoft.cs.rice.edu/open64/ [May 2005].
24. Chapman BM, Huang L, Jost G, Jin H, de Supinski B. Support for flexibility and user control of worksharing in OpenMP. *Technical Report NAS-05-015*, National Aeronautics and Space Administration, October 2005.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe