A COMPILE-TIME OPENMP COST MODEL

A Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Chunhua Liao

August 2007

A COMPILE-TIME OPENMP COST MODEL

Chunhua Liao

APPROVED:

Dr. Barbara Chapman, Chairman
Dept. of Computer Science

Dr. Christoph Eick
Dept. of Computer Science

Dr. Olin Johnson
Dept. of Computer Science

Dr. Yuriy Fofanov
Dept. of Computer Science

Dr. James M. Briggs
Dept. of Biology and Biochemistry

Dean, College of Natural Sciences and Mathematics

Acknowledgements

First and foremost, I would like to thank my advisor, Dr. Barbara Chapman, for her advice, encouragement and support throughout my Ph.D. study. It has been a privilege for me to work with her.

I always feel very lucky to work in a friendly research environment with great people. My thanks go to Dr. Tien-Hsiung Weng, Dr. Zhenying Liu, Dr. Lei Huang, Oscar Hernandez, Laksono Adhianto and Yi Wen.

I must thank my dissertation committee for their comments about my research. They are Dr. Christoph Eick, Dr. Olin Johnson, Dr. Yuriy Fofanov, and Dr. James M. Briggs.

My internship at Lawrence Livermore National Laboratory has greatly deepened my understanding in OpenMP language and compilers in general. I really appreciate my mentor, Dr. Dan Quinlan, for offering the invaluable opportunity in this prestigious laboratory.

I could never have done this without my family's unconditional support. All I have done is dedicated to them: my wife, my parents and my parents-in-law.

A COMPILE-TIME OPENMP COST MODEL

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Chunhua Liao

August 2007

Abstract

OpenMP is a *de facto* API for parallel programming in C/C++ and Fortran on shared memory and distributed shared memory platforms. It is also being increasingly used with MPI to form a hybrid programming model and is expected to be a promising candidate to exploit emerging multicore architectures.

An OpenMP cost model is an analytical model that reflects the characteristics of OpenMP applications on given platforms and estimates the cost, mostly in cycles, of their execution. Such a model could be widely used to evaluate designs of computer architectures, guide compiler transformations, enhance adaptive runtime support, and assist advanced performance analysis. However, existing cost models for OpenMP make over-simplifying assumptions and ignore many software and hardware details. They are too simplistic to support significant optimizations, or meet the demands that are likely to be placed on them in the near future.

In this dissertation, we propose a novel OpenMP cost model to consider OpenMP language details along with application features and platform profiles. A prototype has been implemented in an optimizing and portable OpenMP compiler. The experimental results show that our OpenMP cost model can provide relatively accurate estimations. Moreover, we evaluated the performance characteristics of OpenMP on some multicore and multithreaded architectures and explored possible extensions to our OpenMP cost model for them. We believe our model can meet the demands of different application scenarios and can serve as a basis for a wide range of model-based compilation and runtime systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

OpenMP [86] is a popular parallel application programming interface (API) that extends existing programming languages like C/C++ and Fortran 77/90 to include additional parallel semantics. It consists of a set of compiler directives, runtime library routines and environment variables. Compilers, including corresponding runtime systems, are essential for OpenMP because the code is inserted with high level parallel abstractions that rely on compilers to be converted into low level, explicit multithreaded code for execution. Because of its ease of use, incremental parallelism, performance portability and wide availability, OpenMP is well known in the high performance computing community and has been widely regarded to be a successful parallel programming model on shared memory platforms. It is also being increasingly used with MPI [36] to form a hybrid programming model for additional speedup.

Cost models [68] are a wide class of low level analytical models to reflect the

detailed performance characteristics of computer software and hardware and estimate the cost of executing programs. They are used to evaluate different computer system designs, to select compiler transformations, and also to tune application performance. In particular, optimizing compilers often rely on analytical models of both applications and platforms to guide program transformations. The static estimate of execution cost the models provide helps to find optimal parameters for one optimization phase [77] and/or the best ordering of several optimization phases [112].

An OpenMP cost model [68] is an analytical model to reflect the detailed characteristics of OpenMP applications on given platforms and to estimate the cost, mostly in CPU cycles, of their execution. By providing accurate estimation of execution cycles of OpenMP applications (or portions of them), the model can be widely used in many different scenarios. For example, in an OpenMP compiler, the workload of a parallel loop might need to be estimated first using a cost model to justify costly OpenMP transformations. The parallel loop might be executed sequentially if there is not enough work in the loop to amortize the overhead of parallel execution. For an adaptive OpenMP runtime system, evaluation of different execution configurations, including those using different numbers of threads and scheduling policies, are required before an optimal one can be chosen. In hybrid MPI/OpenMP applications, the optimal number of threads of the inner OpenMP code for each MPI process cannot be decided without a reliable cost model that can help to achieve load balancing.

As computer components decrease in size and uniprocessor designs reach their limits [73], architects have begun to consider different strategies for exploiting the

space on a chip. A dominant trend is to implement Chip-level MultiThreading (CMT) [102] in the hardware. This term refers to supporting the execution of two or more threads within one chip. It may be implemented through several physical processor cores integrated into one chip (Chip MultiProcessing, CMP or multicore) [84], a conventional uniprocessor with replication of features to maintain the state of multiple threads simultaneously (Simultaneous multithreading, SMT) [107] or the combination of CMP, SMT and hardware supported fine-grain [56] or coarse-grain [29] multithreading technologies.

The emerging multicore, multithreaded architectures make OpenMP more popular and the need for OpenMP cost models even more urgent. As the major player in expressing thread level parallelism, OpenMP is a promising candidate to exploit new multicore, multithreaded processors and is expected to be more widely used than before. It could be the right programming model for CMT platforms if OpenMP compilers could do well. However, due to the diverse resource sharing schemes inside multicore architectures, many factors including thread number, scheduling, synchronization, data set partition, and memory locality maintenance in OpenMP have to be reinvestigated, as demonstrated by many previous studies [63, 11, 31]. An accurate OpenMP cost model can help the OpenMP compilation and runtime systems to address these multicore challenges by providing accurate estimation of execution cycles for various situations. It could also act as an alternative and complementary way to empirical-search based solutions, especially for applications using non-iterative algorithms and/or having large search space.

However, only a very few OpenMP cost models [108, 101] exist today and they

are too simplistic to meet the wide demands in the near future. Traditional OpenMP cost models often make over-simplifying assumptions and ignore many software and hardware details. For instance, they assume perfect linear speedup for a parallel loop without considering the cache effect and load balancing within multiple threads. Among many different types of OpenMP-related overhead, only thread fork-join overhead has been modeled so far. Moreover, the new hardware features of multicore architectures have not been explored for cost models yet. While simplified OpenMP cost models have the benefits of easy implementation and minimum overhead, they cannot distinguish between different variants of the same OpenMP code on new diverse platforms and thus cannot provide sufficient accuracy for many practical usages.

In this dissertation, we propose a novel OpenMP cost model to consider sufficient OpenMP language details along with application features and platform profiles. A prototype has been implemented in an optimizing and portable OpenMP compiler. The experimental results show that our OpenMP cost model can provide relatively accurate estimations for practical usages. Moreover, we explore additional extensions to model multicore architectures based on our comprehensive evaluation of OpenMP on typical multicore and multithreaded platforms. We believe that our model can meet demands of different application scenarios and can serve as a basis for various model-based compilation and runtime systems.

1.1 Contributions

This dissertation has the following contributions: 1) a robust, complete OpenMP compiler using both native compilation and source-to-source translation approaches to facilitate a wide range of OpenMP-related research; 2) a novel OpenMP cost model considering sufficient language details with prototyping implementation using the compiler; 3) a comprehensive evaluation of OpenMP on multicore platforms exposing the new challenges for OpenMP cost models; and 4) an exploration of potential extensions of our OpenMP cost model for multicore architectures.

1.2 Dissertation Organization

The remainder of this dissertation is organized as follows. In the next chapter, OpenMP language and current implementation are introduced. Parallel architectures and their trends are discussed in Chapter 3. Chapter 4 presents an optimizing and portable OpenMP compiler built on top of an open source compiler, Open64 [6]. The motivation, design, implementation and evaluation of an OpenMP cost model are given in Chapter 5. Chapter 6 evaluates OpenMP on multicore platforms and explores several extensions of our OpenMP cost model for them. Finally, Chapter 7 concludes and outlines future work.

Chapter 2

OpenMP

OpenMP [86] is a parallel application programming interface (API) that extends existing programming languages like C/C++ and Fortran 77/90 to include additional shared memory parallel semantics. The extensions OpenMP provides contain 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. This chapter introduces background information about OpenMP and its implementation.

2.1 OpenMP Language

The motivation of OpenMP was essentially a standardization of the last 15 years or so of SMP (Symmetric Multi-Processing) development and practice. For computers with a shared-memory architecture, the ability to use directives that assist the

compiler in the parallelization of application codes has been around for many years. Almost all major manufacturers of high performance shared-memory multiprocessor computers have their own sets of directives. Unfortunately, the functionalities and syntaxes of these directive sets vary among vendors and because of that variance code portability (from the viewpoint of directives) is practically impossible. Primarily driven by the need of both the high performance computing user community and industry to have a standard to ensure code portability across shared-memory platforms, an independent organization, openmp.org, was established in 1996. This organization's charter is to formulate and oversee the establishment and maintenance of the OpenMP standard.

2.1.1 History

OpenMP is a young API. In 1997, the first version of OpenMP for Fortran emerged. Since then, OpenMP Architecture Review Board (ARB) has maintained the OpenMP specification and released several new versions (shown in Figure 2.1) to cover C/C++ languages with more features. Two OpenMP specifications were developed for C/C++ and Fortran versions at the early stage until OpenMP 2.5, in which both versions were merged together into a single specification. Using OpenMP, it is relatively easy to create parallel applications in Fortran, C and C++ on shared and distributed shared memory systems.

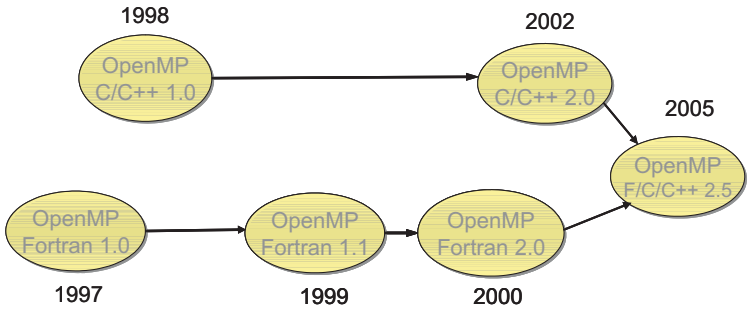

Figure 2.1: Evolution of OpenMP specification

2.1.2 Major Contents

OpenMP adopts a simple fork-join model (presented in Figure 2.2). An OpenMP program begins with an execution using a single thread, called the *master* thread. The master thread spawns teams of threads in response to OpenMP directives, which perform work in parallel. OpenMP also allows the threads to spawn another level of teams of threads inside a parallel region, which is call nested parallelism. Parallelism is thus added incrementally; the serial program evolves into a parallel one. OpenMP directives are inserted at key locations in the source code. These directives take the form of comments in Fortran and `#pragmas` in C and C++. Most OpenMP directives apply to structured blocks, which are blocks of code with one entry point at the top and one exit point at the bottom. The compiler interprets the directives and creates the necessary code to parallelize the indicated code blocks.

Major OpenMP directives enable the program to create a team of threads to execute a specified region of code in parallel (`omp parallel`), the sharing out of work in a loop or in a set of code segments (work-sharing constructs such as loop construct `omp do` in Fortran (or `omp for` in C/C++), sections construct `omp sections`),

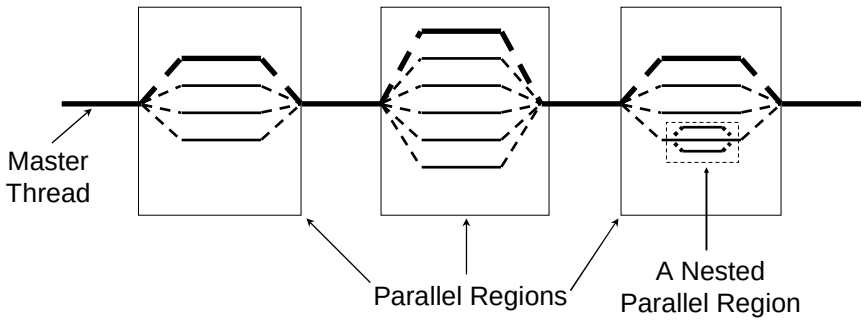

Figure 2.2: Fork-join model of OpenMP

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 runtime behavior of OpenMP applications particularly by setting defaults for the current execution. 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.

For the loop construct (`omp do` in Fortran or `omp for` in C/C++), a `schedule` clause with several variants is provided to specify how iterations of the loop are assigned (scheduled) among threads in order to achieve load balancing. `schedule(static)` evenly divides the iteration space into several chunks and each thread is assigned at most one chunk. `schedule(static,chunk_size)` indicates that the iterations are divided into chunks of the specified size and the chunks are assigned to threads in a round-robin fashion. The way of assigning chunks to threads is fixed for `static`

scheduling once the iteration number and thread number are known. For `dynamic` scheduling, the assignment dynamically happens at runtime. Each thread grabs one chunk of the loop iterations at a time to execute and requests another one when it finishes the current chunk, until no more chunks remain. `guided` scheduling is similar to the `dynamic` scheduling except that the chunk size is proportional to the number of the remaining iterations divided by the number of threads, until reaching to the specified chunk size. Finally, the choice of the scheduling policies mentioned above can be deferred until runtime using an environment variable `OMP_SCHEDULE` if `schedule(runtime)` is used. Figure 2.3 exemplifies the different scheduling policies when 3 threads are used to execute a loop with 18 iterations.

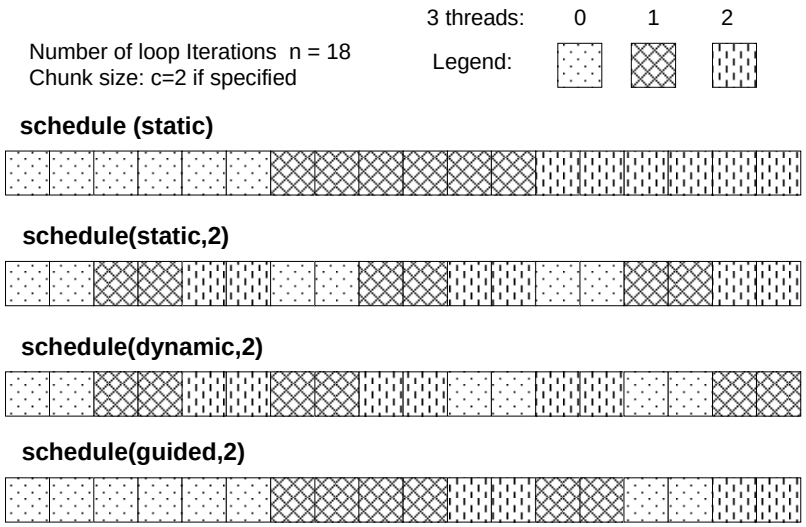

Figure 2.3: Loop iteration scheduling in OpenMP

2.1.3 Examples

Figure 2.4 lists an OpenMP version of the `Hello World!` program, which prints out "Hello world!" text in multiple threads along with their corresponding thread IDs. The parallel construct `omp parallel` is used to enclose a parallel region containing the `printf` statement. The runtime routine `omp_get_num_threads()` returns the ID of the current thread. Preprocessing directives, `#ifdef` and `#endif` are used to support a conditional compilation of the code by a conventional sequential compiler or an OpenMP compiler. From this example, we can see that: an OpenMP program is very similar to its original sequential version since OpenMP only adds additional parallel semantics into sequential code; writing multithreading programs using OpenMP is easy because the actual thread manipulation is hidden from users.

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif
int main(void)
{
  int i=0;
  #pragma omp parallel
  {
  #ifdef _OPENMP
  i=omp_get_thread_num();
  #endif
  printf("Hello, World! I am thread %d\n.",i);
  }
  return 0;
}
```

Figure 2.4: "Hello World" using OpenMP

A more realistic example is given in Figure 2.5, in which a loop calculating the value of PI is parallelized using OpenMP. In this case, a work-sharing construct

`omp for` is combined with the parallel construct `omp parallel` to indicate that the loop body is going to be executed in parallel and the work of the entire iteration space is shared by multiple threads. A data environment clause `private` is needed to keep a private copy of the temporary variable `x` used for each thread, preventing possible race condition otherwise because the default data attribute for all variables is `shared` in OpenMP. The partial sums of the loop iterations assigned to multiple threads are summarized into a final one using the `reduction` clause using `addition` operation. In summary, it is critical for users to judge if a loop is parallelizable before adding OpenMP directives and to make sure the data environment attributes for each variable is implicitly or explicitly specified as desired.

```
#include <stdio.h>
int main (void)
{
  int i;
  double x, pi, sum = 0.0;
  double step;

  long num_steps = 10000000;
  step = 1.0 / (double) num_steps;
#pragma omp parallel for reduction (+:sum) private (x)
  for (i = 1; i <= num_steps; i++)
    {
      x = (i - 0.5) * step;
      sum = sum + 4.0 / (1.0 + x * x);
    }
  pi = step * sum;
  printf ("step:%e sum:%f PI=%.20f\n", step, sum, pi);
  return 0;
}
```

Figure 2.5: PI calculation using OpenMP

The success of OpenMP relies on several advantages: easy of use, incremental parallelism, uniform code for both sequential and parallel versions, and portability.

However, it also has some limitations: it only runs efficiently on small to middle-scale SMP platforms; compiler and runtime support are needed to compile and execute OpenMP programs; there is only limited support for parallelization program structures other than loops.

2.1.4 The Future of OpenMP

The OpenMP language committee is actively working on the new OpenMP 3.0 specification for enhanced expressivity, better performance and scalability, wider usage on more platforms, as well as sustained simplicity. We list several candidate topics below as the representative cases exemplifying the future of OpenMP. They are in various states of readiness and of different degrees of priorities.

1. The biggest addition is supporting tasking. This addition is based on Intel's proposal of `taskq` [103] to support irregular task-level parallelism, such as the parallelizable tasks expressed using pointer-chasing loop iterations and `while` loop iteration.

2. A new clause `collapse` to allow collapsing of perfectly nested loops to enlarge the outer loop iteration space and to amortize parallelization overhead.

3. Supporting NUMA (non-uniform memory access) platforms. The most important candidate is `migrate_next_touch` directive which is used to move data to the node where the next thread would access during runtime.

4. More versatile loop `schedule(runtime)` clause supporting customized schedule

kinds.

5. Better support for nested parallelism, including allowing calling `omp_set_num_threads()` inside a parallel region, adding new library routines to manipulate depth and IDs of thread hierarchy, and allowing `threadprivate` variables to persist across nested parallel regions.

6. Other enhancements and clarifications like allowing unsigned integers as parallel loop control variables, making it clearer the manner of constructing and destructing private objects, adding an environment variable for runtime systems to set idle treads busy wait or sleep at `barrier` and `lock`s.

2.2 OpenMP Implementation

OpenMP has been implemented in many compilers [95, 18, 106, 104]. An OpenMP compiler typically contains several language frontends recognizing OpenMP directives in different programming languages, a transformation phase to restructure OpenMP code, as well as a runtime library to support the parallel execution. In deed, a large fraction of the work of implementing this API is within the runtime library support. Thus the OpenMP transformation phase converts code with OpenMP directives into corresponding multithreaded code with runtime library calls.

Research compilers [95, 18] are typically light-weight source-to-source translators which convert source code with OpenMP constructs into explicitly multithreaded code making calls to a portable OpenMP runtime library. Then, a backend 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. But, unfortunately, the lack of interactions between the two distinct compilers, which naturally do not share any program analysis, may potentially have a negative impact on achievable performance.

In contrast, most commercial compilers [106, 104] enable OpenMP using a module integrated within their sophisticated optimizing infrastructures. In this case, the OpenMP translation has access to program analysis information and can work tightly with all other phases. 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 reexamined 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. 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 backend infrastructure is seldom available.

Figure 2.6 depicts a high level overview of a generic OpenMP implementation: users develop parallel code using OpenMP API; compiler writers build code transformation to translate OpenMP code into multithreaded code with OpenMP runtime library calls; library writers provide thread manipulation using low level thread API such as POSIX Threads (Pthreads) [21]. More details about the transformation and

runtime support are discussed in the following subsections.

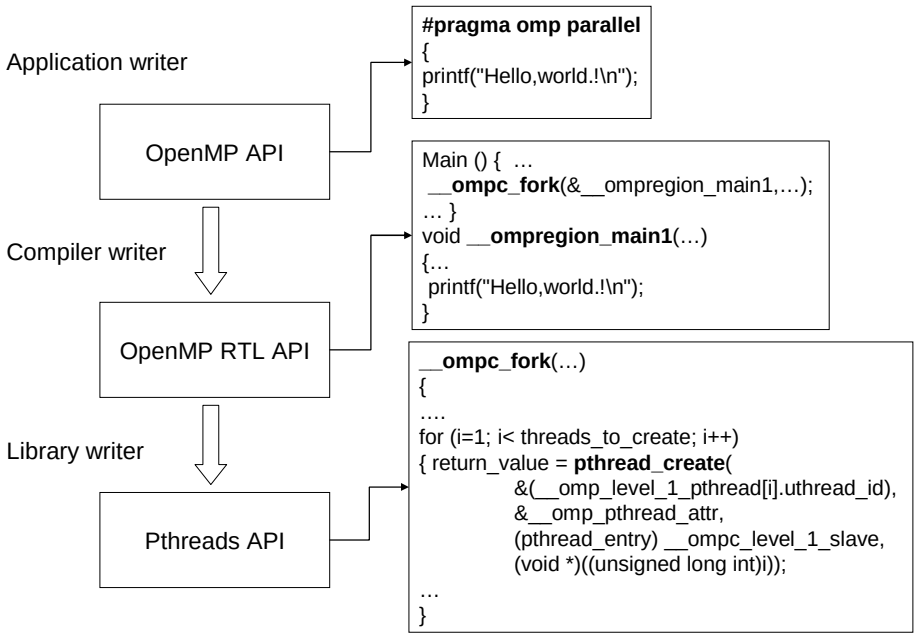

Figure 2.6: OpenMP implementation

2.2.1 OpenMP Translation

A key feature of any OpenMP compiler 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 [95] and OdinMP/CCp [18]. 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 to contain 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, variables that are to be shared among worker threads

have to be passed as arguments to the outlined function. Unfortunately, this introduces some overhead. 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.

Another translation approach is to generate 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 are able to easily process both the original function and the inlined microtask, thus providing more opportunities for analyses and optimizations than the outlining method. A similar approach named the Multi-Entry Threading (MET) technique [106] is used in Intel's OpenMP compiler.

Figure 2.7 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 one compiler(Omni compiler [95]) differs from the inlining translation in the other (OpenUH compiler [87]).

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

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 2.7: OpenMP translation: outlined vs. inlined

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. After that, all the inlined translation method needs to do 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)`. On the other hand, the outlined translation method has more to take care of, since it must wrap the addresses of shared local 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)`. It is clear from the comparison that the inlined translation leads to shorter code and is more amenable to subsequent compiler optimizations than the outlined translation.

2.2.2 OpenMP Runtime Library

The role of the OpenMP runtime library is at least twofold. First, it must implement the 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 API (to perform tasks such as thread creation, synchronization, suspension and waking-up) and deal with repetitive tasks (such as internal variable bookkeeping, calculation of chunks for each thread used in different scheduling options). The actual thread API could be any choice available on target platforms, including Pthreads and Solaris threads.

The existence of an OpenMP runtime library has many benefits. It can free compiler writers from many tedious chores that arise in OpenMP translation. Library writers can often conduct performance tuning without needing to delve into details of the compiler. However, although all OpenMP runtime libraries are fairly similar in term of the basic functionality, 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. Thus, there is no a single standard OpenMP runtime library that can serve all OpenMP compilers so far.

Chapter 3

Parallel Architectures

This chapter briefly introduces the background of major modern parallel architectures including the recent trend of using chip multithreading in computer architecture designs.

3.1 Introduction

One traditional way to classify computer architectures, proposed by Michael Flynn [35], is based on the presence of single or multiple instruction streams and data streams:

- Single Instruction, Single Data stream (SISD): a sequential uniprocessor computer executing one instruction stream to process one data stream.
- Multiple Instruction, Single Data stream (MISD): a computer with multiple

processors applying different instruction streams to a single data stream. This type of architecture is rare since multiple instruction streams generally require multiple data streams to be effective.

- Single Instruction, Multiple Data streams (SIMD): a computer having multiple processors executing the same instruction sequence to process different data streams. It covers vector(array) processors and graphics processing units (GPUs). Small-scale SIMD components such as those supporting Intel's MMX and SSE extensions are also popular inside desktop processors today.

- Multiple Instruction, Multiple Data streams (MIMD):a computer capable of simultaneously executing multiple instruction streams on different data streams. Most modern multiprocessor parallel computers belong to this category.

We focus on MIMD parallel architectures only in this chapter because of their dominant popularity. Depending on the views of the memory subsystems, MIMD parallel architectures can be further divided into the following two categories: distributed memory architectures and shared memory architectures.

3.2 Distributed Memory Architectures

A distributed memory architecture connects multiple autonomous processors with an interconnection network and each of the connected processors has its own private memory. Data in one processor's private memory is not directly accessible to the others and explicit message passing is needed if some data need to be shared. Figure

3.1 shows a typical distributed memory system using a bus-like network. Other interconnection network topologies including ring, mesh, tree and hypercube are also used to implement distributed memory architectures.

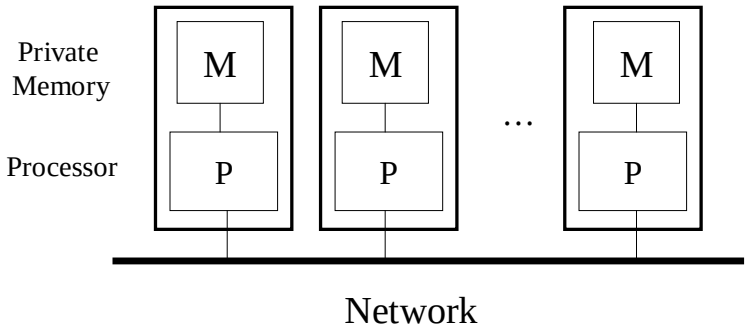

Figure 3.1: A distributed memory architecture

Distributed memory architectures can be easily scaled up to hundreds or even thousands of processors using fast networking. It is also relatively cheap to build because commodity off-the-shelf workstations and networking products can be directly used with flexible configurations. In the list of the top 500 supercomputers [7] released in November 2006, more than 70% ranked computers are distributed memory systems using cost-effective clusters in which workstations are connected by local area networks. For example, one of the largest distributed memory systems, Thunderbird Linux Cluster installed in NNSA[1]/Sandia National Laboratories, has at least 4,096 Dell PowerEdge 1850 nodes connected by a fast Infiniband connection with less than 5.0 micro-second MPI latency and 1.8 GB/s Bandwidth. Each node has dual Intel Xeon EM64T 3.6 GHz processors with 6GB RAM and 73GB SCSI hard drive. Its peak performance reaches 64,972.8 GFlops.

One major disadvantage of distributed memory architectures is that programming

[1]National Nuclear Security Administration

using explicit message passing such as MPI [36] is very difficult for users. Significant efforts are needed to synchronize the work and interchange the data among processors. Consequently, the resulting code is hard to maintain and debug.

3.3 Shared Memory Architectures

In shared memory architectures, all processors have access to the same shared memory space. The machines using shared memory architectures are often called shared memory multiprocessor systems or symmetric multiprocessing (SMP). Figure 3.2 illustrates a typical SMP in which multiple identical processors connect to a single shared memory. Cache coherence among multiple processors is necessary because non-cache-coherent SMPs are prohibitively complex to program.

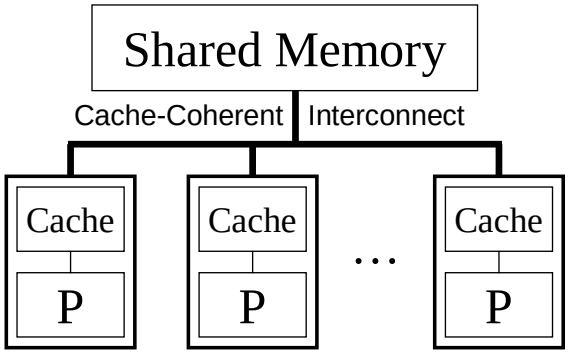

Figure 3.2: Symmetric multi-processing (SMP)

A prominent benefit of SMPs is that modifications to the memory by one processor are visible to others, no explicit message passing needed. Therefore, programming models such as OpenMP for SMPs are much simpler compared to those for

distributed memory architectures. However, the connection between multiple processors and a shared memory can easily become a bottleneck when the number of processors is large. Another complication is that sophisticated hardware support is needed to maintain the cache coherence. As a result, most SMPs are only implemented using up to 32 processors.

To build large scale shared memory systems with hundreds of processors, architects connect a set of small-scale multiprocessor-memory modules and emulate a shared memory space on top of them. This type of systems have non-uniform memory access (NUMA) depending on the distance from one processor to one memory chip. Again, cache coherence has to be maintained among all processors to simplify the programming model. So most NUMA systems are also cache coherent NUMA or ccNUMA. A typical example, SGI Origin 2000, is shown in Figure 3.3.

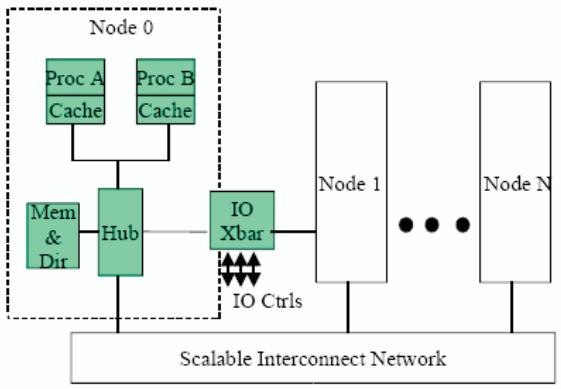

Figure 3.3: SGI Origin 2000, taken from [24]

ccNUMA architectures can scale fairly well. One of the largest ccNUMA machines, SGI Altix 3700, has 512 Itanium2 processors with 1 terabytes memory. Twenty SGI Altix 3700 nodes have been connected and installed in NASA Ames

Research Center to form a super cluster named Columbia supercomputer, that was ranked 8th in the Top500 list in November 2006 [7]. Similar to regular SMPs, simple programming models including OpenMP can be directly used on ccNUMA machines due to their emulated shared memory space. However, getting scalable performance on ccNUMA machines demands careful control of data placement to minimize costly remote memory accesses.

3.4 Chip Multithreading and Multicore

As computer components decrease in size, architects have begun to consider different strategies for exploiting the space on a chip. A dominant trend is to implement Chip-level MultiThreading (CMT) [102] in the hardware. This term refers to supporting the execution of two or more threads within one chip. It may be implemented through several physical processor cores integrated into one chip (Chip MultiProcessing, CMP or multicore) [84], a conventional uniprocessor with replication of features to maintain the state of multiple threads simultaneously (Simultaneous multithreading, SMT) [107] or the combination of CMP, SMT and hardware supported fine-grain [56] or coarse-grain [29] multithreading technologies. We present the reasons for adopting CMT and details of different CMT technologies in this section.

3.4.1 The Limitations of Uniprocessors

Modern superscalar uniprocessors are featured with sophisticated techniques to exploit instruction level parallelism (ILP), such as instruction pipelining, multiple instruction issue per cycle, out-of-order execution, speculative execution, and VLIW (Very Long Instruction Word). Since most codes have low levels of exploitable instruction level parallelism [50] and it is difficult and costly to develop more than 4-issue superscalar pipelines, the performance of uniprocessors has been largely driven by the CPU frequency in recent years.

However, as the size of transistors decreases radically and with the advent of 2 to 3 GHz processors, relentlessly raising the clock speed to enhance processor's performance has reaches its limits. Some challenges include the manufacturing difficulties at nano scale from chemistry and physics points of view, the disproportionately increased power consumption due to current leak, and heat dissipation problems [73]. Architects have been forced to provide alternatives to improve the performance of processors. One dominant trend today is to exploit thread level parallelism (TLP or multithreading).

3.4.2 Multithreading

Multithreading has been realized for a long time by software and hardware approaches. Superscalar uniprocessors can only execute instructions from one thread at a time. So they require a software context switch to execute multiple threads

via time-slicing with high overhead. Early multithreaded processors use hardware-assisted context switch to support multiple threads. They include fine-grain multithreaded processors that switch thread context each cycle [42, 97, 12] and coarse-grain multithreaded ones where a context switch only happens on some long-latency events like remote cache misses [2, 59]. Those early hardware multitheaded processors did not prevail due to the high manufacturing costs. As more transistors are integrated into a single chip with low cost nowadays, various chip level multithreading approaches becomes increasingly popular, including fine-grain/coarse-grain multithreading, simultaneous multithreading, chip multiprocessing and their various combinations.

Simultaneous multithreading [107], also referred to as HyperThreading [57] by Intel, combines hardware multithreading with superscalar processor technology to maximize the utilization of multiple function units of uniprocessors. It allows instructions from several independent threads to be issued within a superscalar processor each cycle. SMT permits all thread contexts to simultaneously compete for and share processor resources. Figure 3.4-(b) sketches the Intel Xeon HyperThreading (Xeon-HT) implementation of SMT. Xeon-HT makes a single physical processor appear as two logical processors: Most physical execution resources including all 3 levels of caches, execution units, control logic and buses are shared between the two logical processors, whereas state resources such as general-purpose registers are duplicated to permit concurrent execution of two threads. Since the vast majority of micro-architecture resources are shared, the additional hardware consumes less than 5% of the die area.

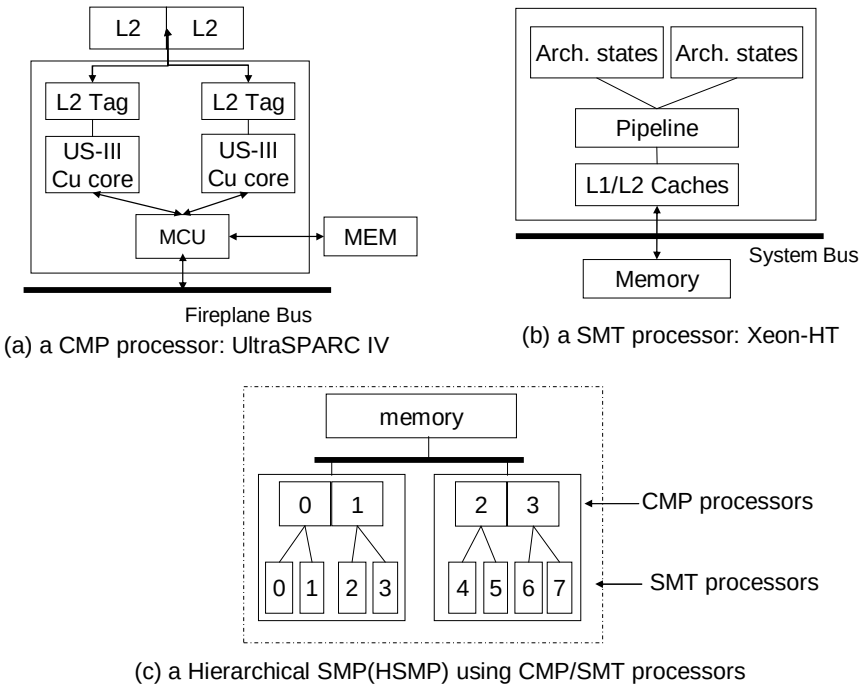

Figure 3.4: CMP, SMT and a hierarchical SMP(HSMP)

Figure 3.5 compares the instruction issue of superscalar processors with those from fine-grain, coarse-grain, and simultaneous multithreading processors. From the figure, it is obvious that the utilization of issue slots in superscalar is very low because of the instruction dependencies and long-latencies operations in the pipeline. The wasted issue slots in one cycle is referred to as horizontal waste(1 to 3 issues/cycle in a 4-issue pipeline) while the 0 issue for some cycles is called vertical waste. A fine-grain multithreaded processor interleaves the instruction issues from several threads, which eliminates the vertical waste of issue slots. The price is some overhead of a hardware thread context switch that happens after every cycle. To avoid excessive context switches, the issue stage of a coarse-grain multithreaded processor does not switch to another thread until a particular long-latency event happens, such as a L2

cache miss. Neither of the fine-grain and coarse-grain multithreaded processors can resolve the horizontal waste of issue slots because only instructions from the same thread can be issued per cycle in both cases. To address this problem, a simultaneous multithreading processor issues instructions from different threads in every cycle to eliminate horizontal waste and achieve maximum utilization of issue slots.

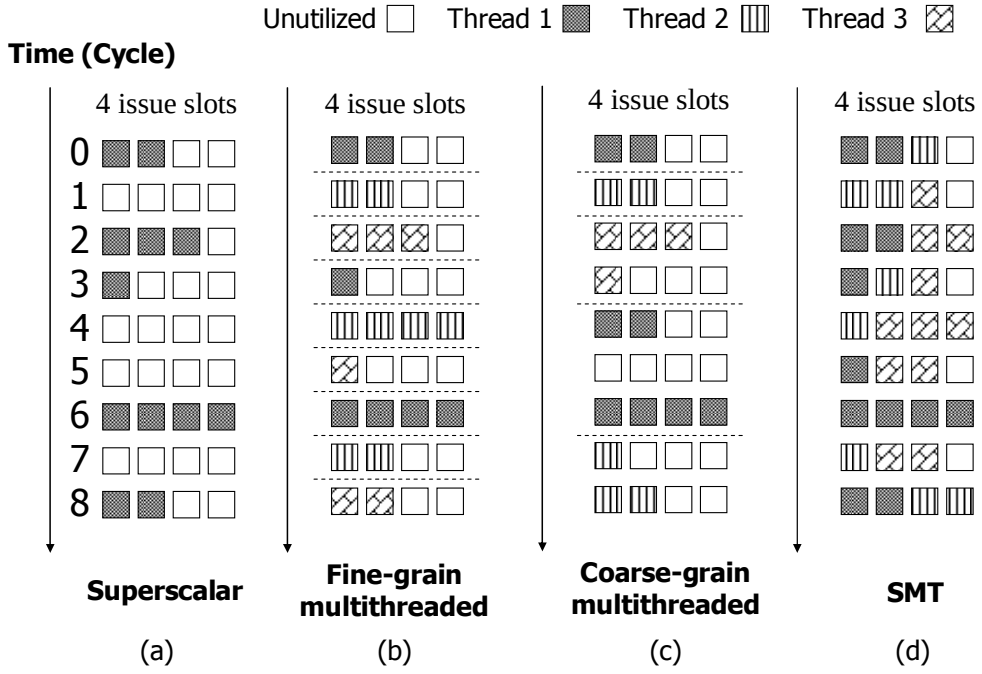

Figure 3.5: Comparison of superscalar and multithreading processors

Chip multiprocessing (CMP or multicore) [84] enables multiple threads to be executed within one chip by integrating several processor cores within one chip. Each processor core has its own resources (pipeline, L1 cache, etc.) as well as shared ones (e.g. off-chip data path). The extent of resource sharing varies from one implementation to another. For example, the UltraSPARC IV [80]'s two cores are almost completely independent except for the shared off-chip data path while the POWER4

[105] processor has two cores with shared L2 cache to facilitate fast inter-chip communication between threads. Figure 3.4-(a) depicts the UltraSPARC IV dual-core processor [46] that was derived from earlier 14-stage, 4-way superscalar uniprocessor designs (UltraSPARC III). Each core has its own private L1 cache and exclusive access to half of an off-chip 16MB L2 cache. The L1 cache has 64KB for data and 32K for instructions. L2 cache tags and a memory controller are integrated into the chip for faster access.

Multicore technology (or CMP) can be used alone [3] or combined with fine-grain [56], coarse-grain multithreading [29] or simultaneous multithreading [51] to form more complex computation and memory hierarchy. CMP combined with SMT and other multithreading features may also be configured in an SMP (we refer to these collectively as hierarchical SMPs or HSMPs below, as shown in Figure 3.4-(c)). While most multicore processors contain identical cores, heterogeneous multicore processors have been seen like CELL [90] where one core is used for generic purposes and the other eight cores are SIMD units.

With great diversity, multicore processors are becoming mainstream for servers, desktop and embedded systems nowadays. Major multicore/multithreaded server processors include IBM POWER4 [105], POWER5, Sun UltraSPARC IV [46], UltraSPARC T1 [56], AMD Opteron, and Intel Itanium 2 dual-core [29]. The competition for the desktop market is more intense. AMD has began shipping dual-core Athlon 64X2 processors since 2005, immediately followed by Intel's release of dual-core Xeon, Pentium D, and Core Duo, Core 2 Duo processors.

In the foreseeable near future, more cores will be integrated into one processor

with even more functionality, greater flexibility, and advanced power management. Intel's Quad-core MCM(Multi-Chip Module) processors have been available since Winter 2006 and eight-core MCM are under test. AMD also plans to produce 8-core processors by 2008. SUN's UltraSPARC T2 is expected to be released by 2007 with 8 cores, each of them handling 8 threads concurrently. For the next decade, Intel envisions a platform with tens of cores, potentially even hundreds of cores per processor. For more efficient computing, special-purpose computation cores for multimedia, recognition and communications will be pervasive. Faster interconnects will link processor cores with new topologies. Caches will be reconfigurable to be globally shared by all cores or exclusively accessible by only a group of cores. The memory subsystem is expected to be on-chip and close to the cores. A single multicore machine might be virtualized to several unified machines to hide the underneath complexity and diversity.

3.5 Summary

Parallel architectures are becoming the mainstream as more transistors will be available in an integrated circuit. However, the advance of hardware is just one of many factors that are necessary to bring more end-user performance. System software including operating systems and compilers, along with different parallel programming models such as MPI and OpenMP, are expected to provide close and innovative support on top of the hardware for developing, running, debugging, tuning and

maintaining new parallel applications. Understanding how to allow legacy sequential code to benefit from parallel machines is another grand challenge today. Even performance alone is not the entire goal. Productivity in human programming and efficiency in energy consumption are equally important concerns to enable scalable and sustainable advances of computing in the long run.

Chapter 4

The OpenUH Compiler

We have designed and implemented an optimizing and portable OpenMP compiler, OpenUH [62], to facilitate our OpenMP-related research, including those for cost modeling. We present the design, implementation and evaluation of OpenUH in this chapter.

4.1 The Design

OpenUH is designed to be a stable, optimizing and portable OpenMP reference compiler. A hybrid approach is adopted in our design, 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 research projects in a setting that is typical of industrial compilers. Given the high cost of building 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 [6] compiler suite, which was open sourced by Silicon Graphics Inc. and is now mostly maintained by a user community. It targets Itanium platforms (Support for X86_64 and ia32 just became available in June 2007). 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 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. Open64 has two IR-to-source translators named *whirl2c* and *whirl2f* which were originally designed to help debugging the compiler at high level IR. Open64 has a partial implementation for OpenMP, but not completely as to the OpenMP V 2.0 specification. It can only handle Fortran OpenMP program until linking phase because there is no corresponding OpenMP runtime library for Open64.

Figure 4.1 outlines an overview of the design of OpenUH based on Open64. It consists of the frontends, optimization modules such as IPA, LNO and WOPT, OpenMP transformation module including `OMP_PRELOWER` and `LOWER_MP`, a portable OpenMP runtime library, a code generator and IR-to-source tools. OpenUH is designed to be 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.

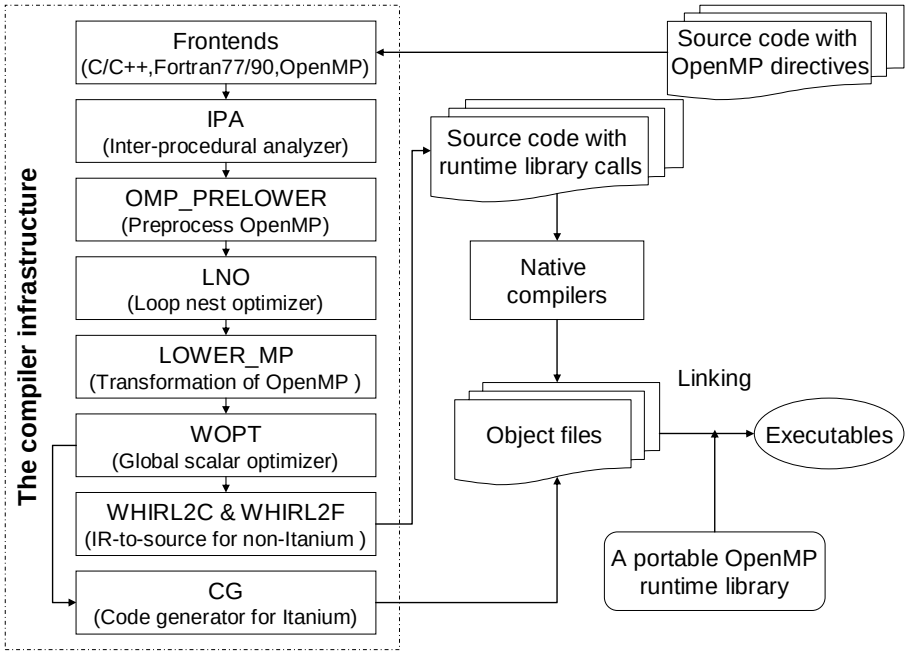

Figure 4.1: The OpenUH compiler

The transformation of OpenMP, which lowers WHIRL with OpenMP pragmas into WHIRL representing multithreaded 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 though the default choice is after LNO as shown in Figure 4.1. 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 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, multithreaded 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. In both native and source-to-source compilation models, a portable OpenMP runtime library is used to support the execution of OpenMP programs. The *whirl2c*/*whirl2f* translators are enabled right before the code generation in order to enable us to apply optimizations at all of the preceding levels, preserving most generic optimizations in IPA, LNO and WOPT, with the cost of harder IR-to-source translation work on Mid level WHIRL instead of the original High Level WHIRL.

4.2 The Implementation

Despite the good match between Open64 infrastructure and our design goals, we faced many practical challenges during the implementation: There was no support for OpenMP directives in the C/C++ frontend, 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 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.

The ORC-OpenMP [27] compiler from Tsinghua University tackled some of the open problems by extending Open64's C frontend to parse OpenMP constructs and by providing a tentative runtime library. At the University of Houston, we 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, including QLogic's commercial-quality PathScale EKO compiler suite [88] targeting AMD x86_64 platforms and the Berkeley UPC [22] compiler [26] with an enhanced *whirl2c* translator.

Based on our design and the initial status of Open64, we focused our efforts on

developing or enhancing four major components of Open64: 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.

To improve the stability of our frontends 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.

4.2.1 OpenMP Translation

OpenUH chooses the nested (or inlined) translation strategy as discussed in Section 2.2.1. In it, the compiler generates a microtask to encapsulate the code lexically contained within a parallel region, and the microtask is nested into the original function containing that parallel region.

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. 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 simple translations; 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 semantics.

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.

Figure 4.2 shows how a parallel region is translated in OpenUH. 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 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.

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 4.2: Code reconstruction to translate a parallel region

Figure 4.3 shows how a code segment containing the work-sharing 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 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 work-sharing construct is also made explicit at the end of the microtask as required by the OpenMP specifications. Chen *et al.* [27] describe in more detail the classification of OpenMP directives and their corresponding transformation methods in the Open64 compiler.

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 4.3: Code reconstruction to translate an `OMP FOR`

Data environment handling is simplified by the adoption of nested microtasks

instead of outlined functions to represent parallel regions. Global variables and local variables in the original functions are visible to the nested microtasks; 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 in 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. `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.

4.2.2 A Portable OpenMP Runtime Library

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 Pthreads API to manipulate underlying

threads as well as to achieve portability.

A major task of the runtime library is to create threads in a team and assign microtasks to them. 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 slave threads and initialize internal variables related to the thread team. The internal variables are used to record things like the number of threads and the default scheduling method. As shown in Figure 4.4, the slave 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 slave threads go back to sleep after finishing their microtask and will wait until they are notified of the next microtask. In this way, the slave threads are reused throughout the execution of the entire program and the overhead of thread creation is reduced to a minimum. The runtime library also support nested parallelism by providing the second level (nested) of slave threads. However, the nested slave threads are not kept alive because they are rarely used.

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. `copyin` is implemented via binary copy from the global value of a `threadprivate` variable to the current

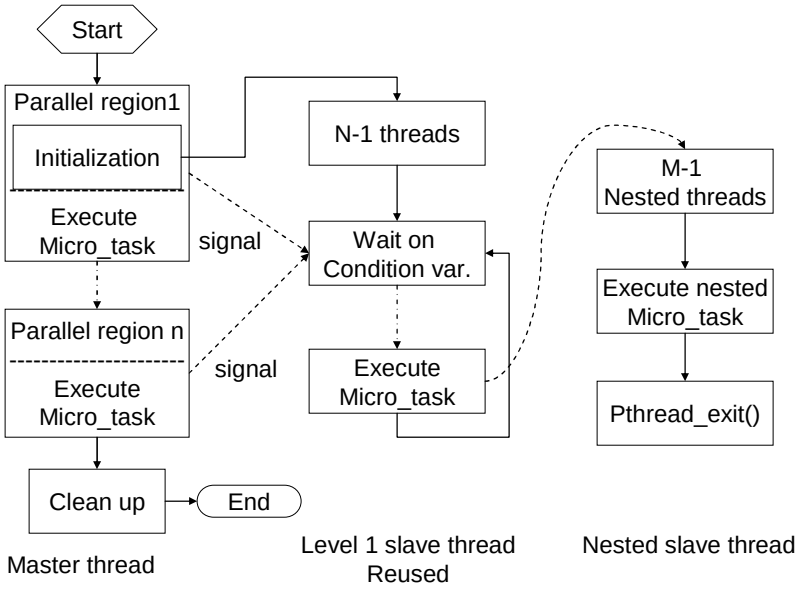

Figure 4.4: OpenUH's OpenMP runtime library

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.

4.2.3 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 analysis 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 (procedures or functions) one at a time and does not treat these in a special way; this will not compile correctly by a backend compiler, since in particular, the shared local variables will become 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 was 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 merged the enhanced *whirl2c* tool from the Berkeley UPC compiler [26] into OpenUH to help resolve some of these problems.

4.3 Evaluation

We have chosen a set of benchmarks and platforms to evaluate the compiler for correctness, performance and portability. The major platform used for testing is COBALT, an SGI Altix system at NCSA (National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign). Cobalt is a ccNUMA platform with a total of 32 1.5 GHz Itanium-2 processors and 256 GB memory. Two other platforms were also used: an IA-32 system running Red Hat 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 a public OpenMP validation suite

[81] 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 overhead of the OpenUH compiler translation of OpenMP constructs. The EPCC microbenchmark [19] has been used for this purpose. Figure 4.5 and Figure 4.6 show the parallel overhead and scheduling overhead, respectively, of our compiler on 1 to 8 threads on COBALT. Compared to the overhead diagrams shown in [19], all constructs have acceptable overhead except for `reduction`, which uses a `critical` section to protect the reduction operation on local values for each thread to obtain portability.

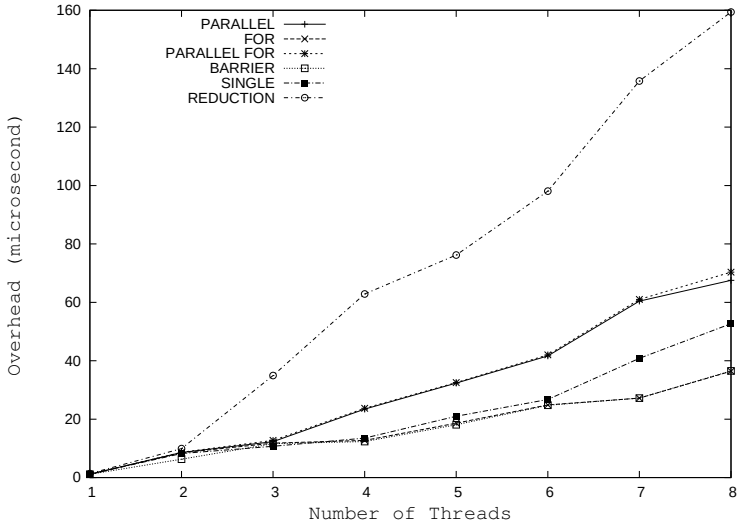

Figure 4.5: Parallel overhead of OpenUH

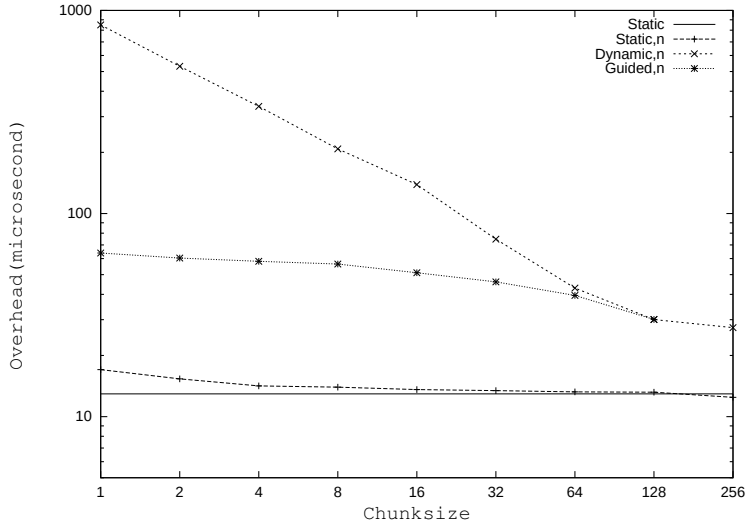

Figure 4.6: Scheduling overhead of OpenUH

We used the popular NAS parallel benchmark (NPB) [48] 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 1 to 16 threads on COBALT. Figure 4.7 shows the normalized Mop/s ratio for seven benchmarks using 8 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 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.

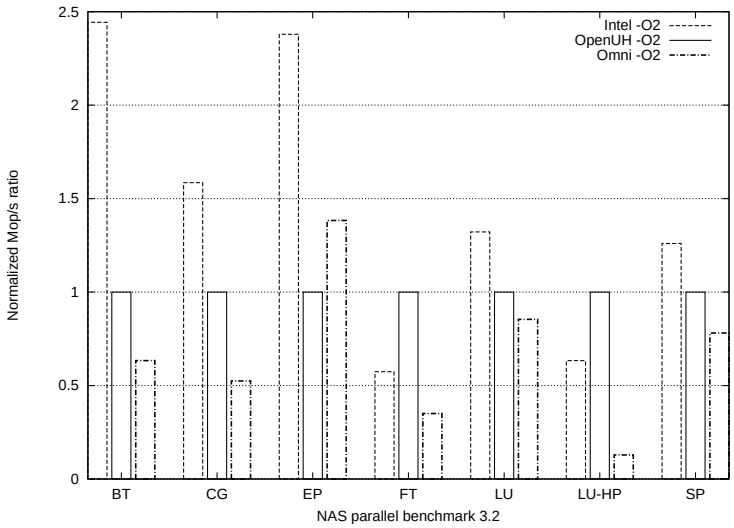

Figure 4.7: Performance comparison of several compilers

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. Also, the Omni compiler and two native commercial compilers (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 4 threads.

Figure 4.8 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 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 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, though 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, thirty percent higher than the source-to-source compilation strategy with `O3` on 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.

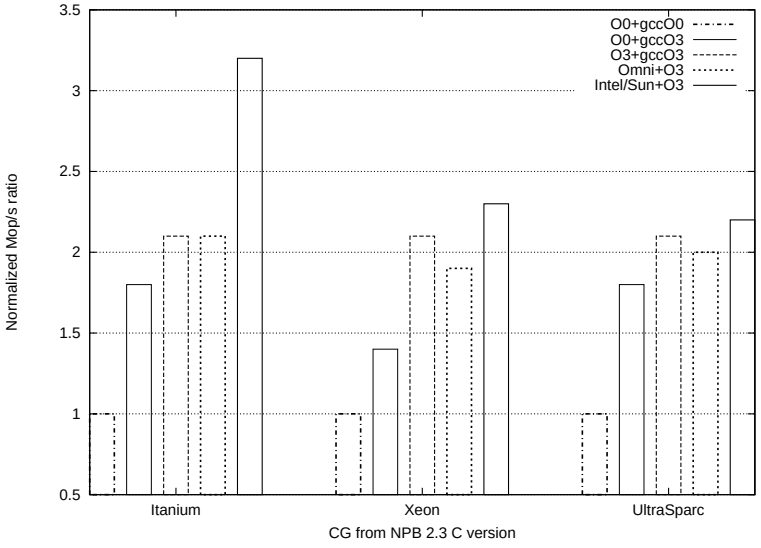

Figure 4.8: Using `whirl2c` with optimizations

4.4 Related Work

Almost all major commercial compilers support OpenMP today. Most target specific platforms for competitive performance. They include Sun Studio, Intel compiler [106], QLogic's PathScale EKO compiler suite [88] and Microsoft Visual Studio. Most are of limited usage for public research. PathScale's EKO compiler suite is open source because it is derived from the GPL'ed Open64 compiler. It is a good reference OpenMP implementation. However, it targets the AMD X8664 platform and its OpenMP runtime library is proprietary.

Several OpenMP research or open source compilers also exist. Omni [95] 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 [18] is another source-to-source translator with only C language support. NanosCompiler [14] 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 a fully functional OpenMP compiler and the source is not released. The ORC-OpenMP compiler [27] 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 only. PCOMP [79] 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. NANOS Mercurium [17], another source-to-source translator for C and Fortran, explores template-based OpenMP translation. Finally, GCC [38] has support for OpenMP since its 4.2 release based on the GOMP project [4].

Besides OpenUH, several other compilers [5, 85, 26] have used Open64 as the basic infrastructure for research goals in language and compiler research. The Kylin compiler [5] is a C compiler retargeting Open64's backend to Intel's XScale architecture. Rice University [85] implements Co-Array Fortran based on Open64. The Berkeley UPC compiler effort [26] uses a similar IR-to-source translation approach to ours to support portable UPC.

4.5 Future Work

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 [25]. Enhancing existing compiler optimizations to improve OpenMP performance on new chip multithreading architectures is also a focus of our investigation [63]. Meanwhile, we are considering an adaptive scheduler to improve the scalability of OpenMP on large scale NUMA systems.

Chapter 5

An OpenMP Cost Model

In this chapter, we present a detailed cost model for OpenMP. Our work is motivated by its important uses in various scenarios and the lack of good OpenMP cost models today. Compared to previous models, our model considers more OpenMP performance factors including cache effects, load balancing, and scheduling policies. We have implemented a prototype of our OpenMP cost model in OpenUH and evaluated it using some benchmarks.

5.1 Introduction

We give a brief overview of cost models first and then discuss the opportunities and challenges for OpenMP cost models in this section.

5.1.1 Cost Models

Cost models [68] are a wide class of low level analytical models to reflect the detailed performance characteristics of computer software and hardware and estimate the cost of executing programs. Various types of cost models exist depending on different criteria. The modeled hardware ranges from one individual computer components such as pipeline, cache and memory to networked supercomputers. The input of cost models, in the form of either source code or machine instructions, may come from basic blocks, loops, procedures, or even whole sequential and parallel applications. The output of cost models could be any performance metrics including execution cycles, cache misses, required memory space, code size, energy consumption and so on. Cost models are being widely used to evaluate different computer system designs, to select compiler transformations, and also to tune application performance. The quality of a cost model is usually judged by its accuracy, time and space complexity, portability, and flexibility.

Optimizing compilers often rely on analytical models of both applications and platforms to guide program transformations. The static estimate of execution cost the models provide helps in finding optimal parameters for one optimization phase [77] and/or the best ordering of optimization phases [112]. Compile-time cost models have many features in common with other standalone performance analysis and prediction models but are unique in that they must combine precision with high efficiency in order to evaluate numerous possible transformations within the limited compilation time. Although high precision of modeled cost compared to measured execution cost (referred to as absolute accuracy) is always favorable for compile-time

cost models, compilers can make correct choices as long as a set of modeled results can correctly reflect the relative soundness of different transformations (relative accuracy).

Figure 5.1 gives an example of a loop tiling transformation in which the original matrix-vector multiplication kernel(shown in Figure 5.1(a)) is transformed into a tiled version (shown in Figure 5.1(a)) using a tile size of $B*B$. Determining whether to tile the loop or not and what is the proper value for the tile size are important decisions for an optimizing compiler. The reason is that the final performance of the loop depends on at least two conflicting factors: cache performance and loop overhead. While tiling might increase the cache utilization, the additional loop overhead introduced by tiling may hurt the final performance. Even if the compiler has decided to tile the loop, the optimal tiling size can not be decided before considering all levels of available caches and their miss penalties. Thus, for the compiler to make the right choice, a low level cost model is needed to consider both cache hierarchy and processor details and to evaluate enough possibilities in advance.

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    c[i] = c[i]+ a[i,j]*b[i];
```

(a) A matrix-vector multiplication kernel

```
for (i=0; i<N; i+=B)
  for (j=0; j<N; j+=B)
    for (ii=i; ii<min(i+B,N); ii++)
      for (jj=j; jj<min(j+B,N); jj++)
        c[ii] =c[ii]+ a[ii,jj]*b[ii];
```

(b) Loop tiling of the kernel

Figure 5.1: An example of loop tiling

Using cost models, compilers can build various model-driven transformations. For example, an algorithm used to find an optimal tiling size for the loop tiling transformation may contain the following steps:

1. Select a new tiling size using some heuristic method;
2. Try titling transformation on the loop with the selected tiling size;
3. Estimate execution cycles of the transformed loop using a cost model;
4. If (stop_condition) goto 5 else goto 1;
5. Stop and choose the transformation with the least cost.

It is well known that building an accurate cost model is a very complex task because numerous factors about software, hardware and their interactions have to be considered. Some typical factors include the application's instruction mixes and their dependencies, memory accesses and reuse patterns, processor's register numbers, function units and their latencies, memory hierarchy and their parameters like access time and miss penalties. Compiler transformations also have significant impact on the final performance. Due to the increasing complexity of applications and hardware and the limited information available at compile-time, versatile, accurate and fast compile-time cost modeling has always been a grand challenge for compiler developers. In practice, pure model-based optimizations are widely regarded to be less successful than empirical tuning [115] except when handling small kernels like the well-studied matrix multiplication. Performance modeling of large scale parallel programs is mostly based on measurement or simulation [69, 44], or some combination

of all available approaches [96] including analytical cost modeling.

As to the accuracy, cost models relying on profiling or benchmarking [94, 98] usually have good average accuracy: the difference ratio between modeled and measured results $((modeled_c - measured_c)/measured_c)$ ranges from $\pm 10\%$ to $\pm 30\%$. Models [49, 9, 23] using static analysis extensively either have much wider difference ranges (from a few percentage to even hundreds percentage) or no accuracy information [110] was provided at all. Most compile-time cost models [113, 76] have not been directly validated for their absolute accuracy. Performance improvement using model-driven optimizations is given, instead.

5.1.2 OpenMP Cost Models

An OpenMP cost model is a cost model using OpenMP applications as its input. Like cost models for sequential programs, OpenMP cost models are useful in many scenarios. For example, the transformation of a parallel loop is usually costly during compilation. An OpenMP compiler may choose to judge the profitability of the parallelization for the loop before the actual translation, by comparing the estimated execution cycles for the sequential version and the parallelized version. Given the choices between different scheduling policies and different chunk sizes, an adaptive OpenMP runtime system may consult an OpenMP cost model to compare enough choices in order to find an optimal execution configuration for a parallel loop. For hybrid MPI/OpenMP applications, OpenMP is often used to balance the load among different MPI processes. An accurate OpenMP cost model is needed to choose the proper

numbers of threads to be used for each MPI process. Finally, performance analysis and prediction tools for parallel applications are other natural users of OpenMP cost models.

Figure 5.2 depicts the performance trend of an OpenMP application on a cc-NUMA machine. Significant performance improvement can be observed in the figure if the application is executed using 1 to 32 threads(processors). However, the performance gain from additional threads can not be sustained for more than 32 threads.

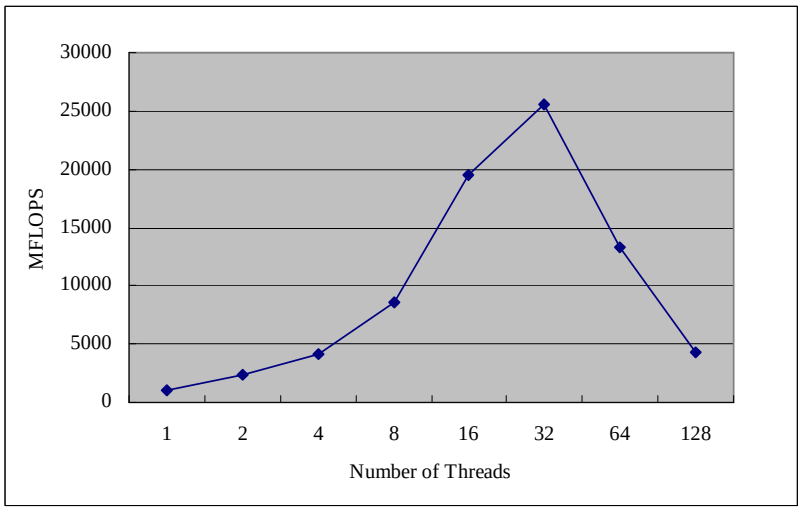

Figure 5.2: Performance of an OpenMP application

Many factors from the application, the OpenMP compiler and runtime system, the operating system, and the machine may contribute to the performance degradation when more than 32 threads are used. An incomplete list of the factors affecting the OpenMP performance is given below:

- The speedup of the application using parallel computing is limited by its non-parallelizable sequential parts as stated by Amdahl's law: speedup $=1/((1-P)+P/S)$, where P is the proportion of the parallelizable parts and S is the number of processors used.

- The workload of the application is not balanced among threads.

- Cheap local memory accesses become costly remote memory accesses when more processors are used to execute the threads in the ccNUMA system.

- The underlying OpenMP runtime library incurs prohibitively high parallel overhead when a large amount of threads are used.

- The operating system wrongfully migrates threads among processors during the execution and causes excessive page faults.

- False sharing of cache lines becomes frequent when more threads have memory requests.

- The memory bandwidth of the shared memory system becomes a bottleneck once more memory requests from multiple processors arrive than the bus can handle.

An ideal OpenMP cost model should consider enough performance factors from software, hardware and their interactions to provide reasonably accurate cost estimation for an application on a given system. At the same time, low overhead of using such a model is desired for frequent uses, especially in a compiler.

However, only a few OpenMP cost models have been proposed so far in the literature. For example, a simple fork-join model [108] is used to derive a threshold for dynamic serialization of OpenMP to avoid unnecessary parallelization. Some simple heuristics [28, 101] for estimating execution time of OpenMP code portions are used to decide the numbers of OpenMP threads to balance MPI load. The previous models can be largely represented using the following equation:

$$T\_OpenMP_c = T\_fork_c + T\_sequential_c/Number\_threads + T\_join_c$$

The equation above only considers the fork-join overhead (T_fork_c and T_join_c) among many OpenMP overheads and assume perfect speedup for the parallelized portion compared to its sequential execution time ($T\_sequential_c/Number\_threads$). In reality, the execution time of OpenMP applications are impacted by more factors such as synchronization overhead, scheduling policies, load balancing, cache effects and so on. Therefore, the previous OpenMP cost models are over-simplifying to meet the wide requirements today.

5.2 Our OpenMP Cost Model

Our OpenMP cost model is aimed to provide trustworthy estimation of execution cycles for OpenMP code by modeling sufficient factors with reasonable overhead. In particular, it has the following goals:

- Modeling multiple work-sharing and synchronization portions in a parallel region,

- Considering more types of OpenMP overhead in addition to the fork-join overhead,
- Distinguishing between different scheduling policies and between different chunk sizes,
- Capturing possible load imbalance among threads,
- Taking cache impacts into account, and
- Having low overhead so that it may be invoked tens or even hundreds of times by a compiler.

To reach our goals, we adopt a divide-and-conquer strategy. A underlying assumption is that all cycles can be classified into independent groups associated with separated factors. As shown in Figure 5.3, we divide the cost of OpenMP execution into two categories: sequential execution cost and parallel execution cost. Each category is modeled by a corresponding sub-model: the sequential sub model and the parallel sub model. The cost of each sub model is further broken down into more fine-grain cost types, which in turn are modeled separately. For example, the sequential execution cost depends on the cost associated with processors, memory and loops; the parallel execution cost is treated as the sum of the costs for executing the parallel region, work-sharing, and synchronization constructs in OpenMP.

The major benefit of the divide-and-conquer strategy is that a modularized design and implementation could be easily achieved. Subsequently, the resulting model should be flexible and extensible. However, the correlation among different factors

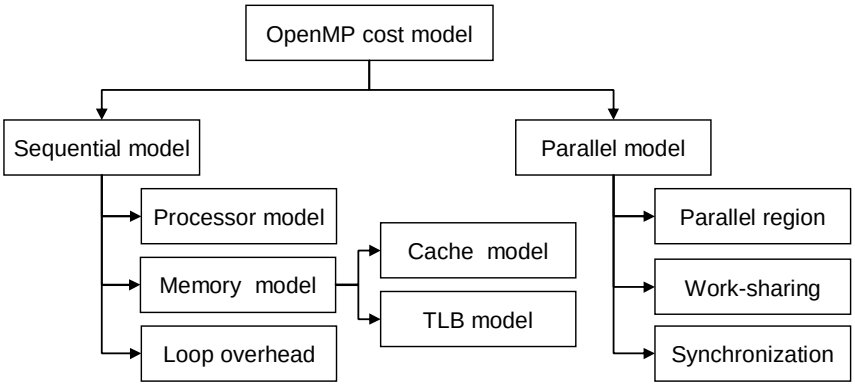

Figure 5.3: An OpenMP cost model

are largely ignored. For example, the processor pipelining cycles might be overlapped with cycles to serve memory requests, and it is hard to attribute some cycles to one individual factor sometimes.

To minimize the overhead of modeling all factors, a combined approach is used to reduce the complexity as well as to maintain its flexibility. For variants in the model, we use analytical modeling to build formulas considering different input values. For invariants, results from direct measuring are applied whenever possible to simplify our model.

As to the input of our model, we choose to use some high level intermediate representation (IR) which preserves OpenMP language features, instead of the low level IR or even machine instructions without OpenMP constructs. The benefit of working on a high level IR is that the high level parallel abstractions expressed in OpenMP can be easily recognized within the modeled applications. However, the accuracy of the output might be compromised because of the difference between the high level IR and its corresponding low level machine instructions after various

compiler transformations.

Figure 5.4 represents our OpenMP cost model in terms of a set of equations. One of them defines the cost in terms of cycles for an OpenMP application ($OpenMP_c$), which is an accumulation of its sequential parts ($Sequential\_Part\_i_c$) and parallelized portions ($Parallel\_Region\_j_c$). The other equations provide users certain derived metrics like parallel efficiency and speedup based on the original sequential execution time ($Sequential_c$) and the OpenMP execution time ($OpenMP_c$).

$$\begin{aligned} OpenMP_c &= \sum_{i=1}^{m} Sequential\_Part\_i_c + \sum_{j=1}^{n} Parallel\_Region\_j_c \\ Speedup &= Sequential_c / OpenMP_c \\ Parallel\_Efficiency &= Speedup / Number\_threads \end{aligned}$$

Figure 5.4: Equations for modeling OpenMP applications

We discuss our OpenMP cost model in more detail in the following subsections.

5.2.1 Modeling Sequential Code

The sub-model for the sequential parts of an OpenMP application is an indispensable component in our OpenMP cost model. On the one hand, it provides the estimation for the execution cost of sequential parts in an application. On the other hand, it is reused to estimate of the cost of executing a chunk of one loop by a single thread in parallel regions. A classic approach is used to model sequential parts in our model as shown in the following equation:

$$Sequential\_Part_c = Processor_c + Memory_c + Loop\_Overhead_c$$

$Processor_c$ represents the execution cycles needed to carry out a piece of sequential code without considering any cache or TLB miss penalties. It depends on the instruction types and mix in the sequential code and the execution resources, such as function units, available in the target processor and their latencies. A compiler can traverse the IR of the input sequential code and map them into low level machine operations such as load/store, arithmetic add and floating point multiplication, etc. A cost table associating the execution cycles for each type of machine operations is needed for the compiler to accumulate the total cycles needed to execute the input code. With all this information ready, the cost model will simulate the instruction scheduling and execution in the processor and accumulate the execution cost based on function unit latency, instruction dependency, register spilling, etc. Detailed information of the processor model will be discussed in Section 5.3.

All cost of memory requests which are not directly served by the L1 cache is considered in $Memory_c$, which in turn may include cache miss penalties from all levels of cache and the page fault penalties from TLB misses. Memory accesses in the sequential code is one of the key inputs for the model, along with the memory hierarchy parameters such as cache size and miss penalties in the target platform. Again, a compiler will traverse the IR of the input code to find out all memory accesses and their reuse pattern and predict misses based on cache configurations of target platforms using various strategies [91, 76, 40]. The cache cost can be simply modeled as a multiplication of the number of misses and average miss penalty. It can be also calculated from different kinds of cache misses including compulsory, capacity, and conflict misses using complex formulas considering cache reuse patterns. It is

up to the implementation to choose the proper way to fit the need based on the trade-off between desired accuracy and affordable complexity. TLB can be modeled using similar methods since it is indeed a special kind of cache.

Loops appear quite often in OpenMP applications and deserve special attention. The total execution time of a loop is not simply the execution cost of one iteration of the loop body multiplied by the iteration count. Additional loop control overhead ($Loop\_Overhead_c$) has to be considered to accurately reflect the execution cost. This overhead comes from the machine instructions to increment the value of the loop control variable, to judge if the condition to iterate the loop body is met, and to jump back to the loop entry. Accurate estimation of loop overhead can be surprisingly challenging because the cost of the jump instruction can potentially be associated with instruction cache misses, data cache misses and branch misprediction cost in the processor pipeline. Also, loop optimizations such as loop unrolling and tiling dramatically change the original loop control structure and make the prediction even more difficult. For simplicity, we assign a constant value as the overhead of one loop iteration. The actual value may derive from direct measuring of the overhead of some simple loop kernels executed on the target platform.

Other concerns when modeling sequential parts include the handling of different possibilities of control flow path and function call chain. Given the limited information available from its static analysis, a compiler usually relies on some simple rules to roughly predict the possible control flow or call chain, though sophisticated inter-procedural analysis could help to some extent. For example, the compiler can assume

that the larger of two `THEN` and `ELSE` blocks will always be executed for a `IF` statement. A more accurate approach is to use the profiling information from previous executions to make the prediction. The implementation may use either approaches depending on the features of applications and tolerable modeling overhead.

5.2.2 Modeling Parallel Regions

According to the OpenMP specification, a parallel region is a portion of code encountered during an instance of the execution of the OpenMP `parallel` construct. Like most other OpenMP cost models, our model considers the thread fork-join overhead and the rest execution time for each parallel region. But we do not assume perfect speedup for a parallel region. Instead, the rest execution time of a region is calculated based on finer-grain models for the execution cost of each thread in the region in order to reflect OpenMP performance in reality. Figure.5.5 lists the equations of a parallel region cost model for C/C++ aspect. The equations for Fortran programs are similar.

$$
\begin{aligned}
Parallel\_region_c &= Fork_c + \sum_{j=1}^{n}[maximum(Thread_0\_exe\_j_c, ..., Thread_{n-1}\_exe\_j_c)] + Join_c \\
Thread_i\_exe\_j_c &= Work\_sharing_c + Synchronization_c \\
Work\_sharing_c &= Omp\_for_c/Omp\_sections_c/Omp\_single_c \\
Synchronization_c &= Master_c/Critical_c/Barrier_c/Atomic_c/Flush_c/Lock_c
\end{aligned}
$$

Figure 5.5: Equations for modeling a parallel region

From Figure.5.5, it is obvious that our model takes multiple work-sharing and synchronization portions into account to reflect all possible usage of an OpenMP parallel region. Another goal of our model is try to capture possible load imbalance

among multiple OpenMP threads. To achieve this, we model the execution time of threads between each pair of synchronization points individually and choose the longest one as the final execution time. The cost of each thread is in turn treated as the sum of encountered work-sharing and synchronization costs. Finally, the execution time of the entire parallel region ($Parallel\_region_c$) depends on the execution time of the most time-consuming thread between each pair of synchronization points($Thread_i\_exe\_j_c$), plus the fork-join overhead.

As to the costs of OpenMP synchronization constructs such as `omp master`, `omp critical` and `barrier` , we choose to use measured results instead of analytical models. The reason is that though synchronization costs highly depend on the OpenMP implementation and the platform, the cost of one particular synchronization construct is usually consistent across applications once the compiler and platform is fixed. There is no need to repetitively model the synchronization cost in the OpenMP cost model. Direct measuring using microbenchmarks in different numbers of threads should be a simpler and more accurate way to get the cost compared to complex modeling for each synchronization construct considering variants of OpenMP implementation and platforms.

5.2.3 Modeling Work-sharing

OpenMP provides a set of work-sharing constructs to distribute the execution of the associated workload among a team of threads that encounters it. Those constructs include `omp for` (or `omp do` in Fortran), `omp sections`, `omp single` and

`omp workshare` (Fortran only). The loop construct (`omp for` or `omp do`) is the most important one because it is frequently used by users and other work-sharing constructs are often converted into it in many OpenMP implementations. Thus, we focus on modeling the loop construct in this subsection.

The loop construct indicates how each thread shares the execution of a loop using some predefined scheduling policies such as `static`, `dynamic` and `guided` scheduling (illustrated in Figure 2.3). Most OpenMP implementations choose to use a set of internal scheduling subroutines enclosed in a `while` loop to perform the chunk generation and dispatching until no more loop iterations are left, which incurs some overhead named scheduling overhead. The scheduling overhead increases as the chunk size decreases since more scheduling operations are needed.

Figure 5.6 shows the equations for modeling the loop construct considering the scheduling overhead cycles ($Schedule_c$) and different chunk sizes. We also distinguish the schedule overhead for different schedule policies. Similar to the equations modeling sequential parts, the execution time of each loop chunk for one thread is calculated from the processor cycles, memory(or cache) cycles and loop overhead. The total execution time of an `omp for` portion of one thread is determined by the scheduling overhead, the cycles needed to execute all chunks assigned to the thread, and possible overhead from `ordered` and `reduction` clauses.

$$
\begin{aligned}
Omp\_for_c &= S\_times * Schedule_c + \sum_{k=1}^{S\_times} Loop\_chunk\_k_c + Ordered_c + Reduction_c \\
Schedule_c &= Schedule\_static_c + Schedule\_dynamic_c + Schedule\_guided_c \\
Loop\_chunk\_k_c &= Machine_c\_per\_iter * Chunk\_size + Cache_c + Loop\_overhead_c
\end{aligned}
$$

Figure 5.6: Equations for modeling an `OMP FOR`

5.3 Implementation

We have implemented a prototype OpenMP cost model in OpenUH. In the following subsections, we describe the existing cost modeling capabilities in OpenUH and our enhancements to implement our OpenMP cost model.

5.3.1 Cost models in OpenUH

OpenUH includes a set of cost models inherited from Open64's loop nest optimizer(LNO) [113] that can be used to estimate, in CPU cycles, the cost of executing singly nested loop (SNL) nests. SNL loop nests comprise perfectly nested loop nests and imperfect loop nests that are eligible to be transformed into perfect ones. The compiler will use its cost models to choose a combination of different loop level optimizations, including loop interchange, tiling and outer loop unrolling. The models are also used to guide automatic parallelization. There are three major models: the Processor (or Machine) model, the Cache model and the Parallel model. We briefly describe each of them below.

The processor model(see Figure 5.7) is mainly used to estimate the CPU cycles needed to execute one iteration of an SNL loop, without considering latencies from the memory hierarchy. The processor cycles ($Machine_c\_per\_iter$) are calculated based on cycles from FP(Floating Point) and ALU units(OP_c), memory units(MEM_ref_c), and issue units($Issue_c$). The compiler uses a preset scheduling table to map High Level WHIRL IR into target machine instructions. The instructions are further mapped to processor resources (FP, ALU, Branch etc.) associated

with latencies. The compiler accumulates cycles for each resource after IR browsing based on the mappings and returns the maximum value as the result. In addition, the compiler counts base registers (reserved ones) and registers used in the loop for scalar and array references to get the total number of registers $Regs\_used$ required in a loop. The spilled registers ($Reg\_used - Target\_Regs$), if they exist, are converted into extra memory references and spilling cycles are included in memory reference cycles.

$$
\begin{aligned}
Machine_c\_per\_iter &= Resource_c + Dependency\_latency_c + Register\_spilling_c \\
Resource_c &= maximum(OP_c, MEM\_ref_c, Issue_c) \\
MEM\_ref_c &= (Num\_fp\_refs + Num\_int\_refs)/Num\_mem\_units \\
Issue_c &= Num\_inst/Issue\_rate \\
Dependency\_latency_c &= maximum(Sum\_of\_latencies_i/Sum\_of\_distances_i) \\
Regs\_used &= Base\_regs + Scalar\_regs + Array\_regs \\
Spilling\_mem\_ref &= (Reg\_used - Target\_Regs) * [Num\_reg\_refs/(Scalar\_regs + Array\_regs)]
\end{aligned}
$$

Figure 5.7: Equations of processor model

The processor model also takes into consideration the latency involved for instructions/memory operation dependencies, which are significant for processor stalls. The assumption here is that the modeled loops will be optimized using software pipelining so that intra-loop dependencies can be ignored. Loops are analyzed to build dependence graphs, in which a vertex represents a floating point load or store and an edge is marked with a latency and an iteration distance. The cycles in the graph are located to get $sum\_of\_latencies_i/sum\_of\_distances_i$. The maximum is returned as $dependency\_latency_c$.

The cache model helps by predicting the cache misses and the associated penalty cycles required to execute inner loops. It creates instruction footprints that represent

the number of bytes of data references in cache by a given instruction. Instruction level footprints are further accumulated to loop level footprints. The unused data residing in the same cache lines as the last referenced data element is also considered as part of the footprints(namely edge effect). To consider spatial data locality, the compiler counts only once for all references to the same array whose index expressions differ by at most a constant in each dimension,(e.g. a[i][j] and a[i][j+1]). Adding the different footprints can predict cache overflow if the sum of the footprints exceeds the cache capacity, which generates cache misses. The final cost (Shown as $Cache_c$ in Figure 5.8) is accumulated from all levels of cache hierarchy using $miss * penalty$ for simplicity. The model distinguishes between read and write cache misses and calculates their cost using clean footprint and dirty footprint, respectively. TLB cycles are calculated using a similar approach as shown in Figure 5.8.

$$
\begin{aligned}
Cache_c &= \sum_{i=1}^{Levels} (Clean\_footprint_i * Clean\_penalty_i + Dirty\_footprint_i * Dirty\_penalty_i) \\
TLB_c &= TLB\_miss\_penalty * TLB\_miss
\end{aligned}
$$

Figure 5.8: Equations of cache model

The parallel model was designed to support automatic parallelization by evaluating the cost involved in parallelizing a loop. It helps to determine if there is sufficient work to justify the parallelization, and if the parallelization is justified, the model is further used to choose the best level of the loop to parallelize taking loop permutations into consideration. The parallel model mainly calculates loop body execution cycles by reusing the processor and cache models, but also includes the parallel overhead from the fork-join operations in the shared memory model, plus the loop overhead. Figure 5.9 shows a high level summary of the equations used

to calculate the total cycles of a parallelized loop from various sources, including processor(machine), cache misses, TLB misses, loop overhead, and parallelization overhead.

$$
\begin{aligned}
Total_c &= Machine_c + TLB_c + Cache_c + Loop\_overhead_c + Parallel\_overhead_c \\
Machine_c &= Machine_c\_per\_iter * Num\_loop\_iter/Num\_threads \\
Loop\_overhead_c &= Loop\_overhead\_per\_iter_c * Num\_loop\_iter/Num\_threads \\
Parallel\_overhead_c &= Parallel\_startup_c + Parallel\_const\_factor_c * Num\_threads
\end{aligned}
$$

Figure 5.9: Equations of parallel model

5.3.2 Modeling OpenMP

We have adapted the parallel model in OpenUH to model OpenMP applications, which involved adding new extensions to the model and modifying its existing implementation.

Adapting the parallel model to model OpenMP has proved to be mostly straightforward. OpenMP uses a simple fork-join execution model and the original parallel model in OpenUH provided a good basis for it, though many important factors were ignored. Our extensions cover more details of the OpenMP programming API and are depicted in Figure 5.4, Figure 5.5 and Figure 5.6.

There are several implementation details which are worthy of mentioning. First of all, the original parallel model only worked on sequential SNL loops. We added a new IR traversal phase to locate all OpenMP parallel regions and to apply the extended cost model to each of them subsequently. Second, the OpenUH compiler is capable of aggressive loop transformations. We chose to place the OpenMP cost model after all

possible loop transformations to allow it to be aware of possible applications of loop interchange, loop unrolling and tiling happening on the parallel loops. Third, most values of the parameters (thread fork overhead, scheduling overhead, etc.) used in the OpenMP model can be obtained or derived from microbenchmarks [20, 100]. Vendor manuals can also provide values of the parameters in the sequential model, especially for the processor and cache. Finally, a lightweight OpenMP scheduler was needed to estimate the chunks of loop iterations assigned to each thread. For simplicity, we use a round-robin chunk assignment method for all scheduling policies including `static`, `dynamic`, and `guided`. It might also be possible to simulate the runtime scheduling inside the OpenMP model based on the estimation of the execution time of each assigned chunk for each thread in the future. But the overhead might be too high for the compiler to explore a large search space.

5.4 Evaluation

We have evaluated the overhead and accuracy, including absolute accuracy and relative accuracy, of our model using a set of benchmarks on a shared memory machine. The methodology and results are given below.

5.4.1 Methodology

Two benchmarks were chosen for the evaluation. One was a classic parallel matrix-matrix multiplication(MMM) kernel (shown in Figure 5.10(a)), which has also been

widely used in previous research [113, 115] due to its importance in scientific numerical computation. The other was a numerical kernel to solve a finite difference discretization of Helmholtz equation using the Jacobi iterative method, detailed in Figure 5.10(b). In both benchmarks, three array sizes (500×500, 1000×1000, and 1500×1500) were used in order to study the impact of different input data sets on our model.

```
#pragma omp parallel for private(i,j,k)
for (i = 0; i < N; i++)
  for (k = 0; k < K; k++)
    for (j = 0; j < M; j++)
      c[i][j]= c[i][j]+a[i][k]*b[k][j];
```

(a) Matrix multiplication in OpenMP

```
for (k=0;k<max_iter;k++)
{
#pragma omp parallel
  {
#pragma omp for private(j)
  for(i=0;i<N;i++)
    for(j=0;j<M;j++)
      uold[i][j] = u[i][j];
#pragma omp for private(j)
  for (i=1;i<(N-1);i++)
    for (j=1;j<(M-1);j++)
     u[i][j] = uold[i-1][j] + uold[i+1][j]\
       + uold[i][j-1] + uold[i][j+1];
  }
}
```

(b) Jacobi kernel

Figure 5.10: Benchmarks to evaluate the model

Our test platform was COBALT, an SGI Altix system at NCSA. COBALT is a ccNUMA platform with a total of 32 1.5 GHz Itanium 2 processors and 256 GB

memory. Major machine, OpenMP and other modeling parameters are shown in Table 5.1 and Table 5.2. The processor parameters are given in terms of available count per CPU cycle such as `Mem_units` and `Branch_units`. The values of the parameters were obtained in several ways: vendor manuals such as Intel's Itanium 2 processor manuals [1], system information from an open source performance analysis toolset named PerfSuite [58], and microbenchmarks such as LMbench [78] and Calibrator [71] for memory hierarchy parameters, and EPCC [20] for OpenMP overhead.

Processor	**Count/Cycle**
Mem_units	4
Issue_rate	6
Float_units	2
Integer_units	6
Branch_units	3
Target_regs	128(int)+128(fp)
Base_regs	10(int)+32(fp)
Other	**Cycles**
Loop_overhead_per_iter	4
Parallel_startup	3000
Parallel_const_factor	500
Schedule_static_c	2000

Table 5.1: Major processor and OpenMP parameters

	Size	**LineSize**	**Clean Penalty**	**Dirty Penalty**	**Assoc.**
L1D	16KB	64	21	21	4
L2	256KB	128	200	200	8
L3	6MB	128	220	220	24
	Entries	**PageSize**			
TLBD	32	16KB	50	50	Full

Table 5.2: Major memory hierarchy parameters

The overhead of our model was measured by dividing the amount of time spent on modeling a parallel region for the first time (*model_time*) by the total compilation

time for the region (*compile_time*). The result can give the upper bound of the overhead because additional modeling of the same parallel region could reuse many intermediate results of the initial modeling. OpenUH's embedded timing utility was used to measure the two metrics during a compilation using the OpenMP modeling.

The absolute accuracy of the model as a whole was evaluated by comparing the modeled results to the corresponding measured results for the benchmarks. Different input data sets and up to 8 threads were used to get a comprehensive and fair comparison. To get the measured results, we used OpenUH to compile the benchmarks with `-mp -O3` because our model assumes that the optimizations occur. PerfSuite [89] was used to collect performance metrics including CPU cycles. OpenUH's internal tracing utility has been extended to dump the final modeled results as well as the intermediate results of sub-models.

Further, we designed a method based on [47] to validate the sub-models for processor, cache and loop cycles. In this method, the measured cycles are broken down into several categories based on values of a set of hardware counters. Two major categories are considered: unstalled cycles and stalled cycles. Stalled cycles can be in turn divided into seven sub-categories such as floating point (FLP) stalls due to floating point register dependencies and floating point function units, stalls to serve L1 data cache (D-cache stalls), branch misprediction stalls, instruction miss stalls, register stack engine (RSE) stalls, stalls from general purpose register dependencies (GR scoreboarding), and CPU frontend flush stalls (FE flushes). Although cycles of the low level categories derived from hardware counters cannot be directly mapped to the cycles of our sub-models, we can roughly assume two rules: D-cache stalls can

be used to evaluate the modeled cache cost; and the sum of unstalled cycles and FLP cycles is a good indication for the sum of processor and loop cycles in our model. The 1000×1000 versions of both MMM and Jacobi kernels using 1 thread were used for the validation.

Finally, the relative accuracy of the model was evaluated via benchmarks using OpenMP scheduling clauses with different chunk sizes. The model's ability to capture the trend of actual cycles needed for various chunk sizes is a good indication for its relative accuracy.

5.4.2 Results

The model has fairly low overhead according to our evaluation. On average, only 1.25% of the total compilation time was needed for to invoke the OpenMP cost model for the first time. Therefore, our model is efficient enough for frequent uses within the compiler.

Figure 5.11 and Figure 5.12 give modeled and measured CPU cycles of the MMM kernel and Jacobi respectively. Mostly, the modeled results have the same trend as the measured ones. It is noticeable that the dashed lines representing the modeled results are all smooth while the solid lines for the measured results have irregular fluctuations sometimes due to system noises, especially for executions using nearly 8 threads to operate on a small data set. Jacobi's results of using 4 to 8 threads are not shown because the measured cycles dramatically increased due to the costs of remote memory accesses to shared boundary array columns. Our model does not

consider such a ccNUMA feature yet.

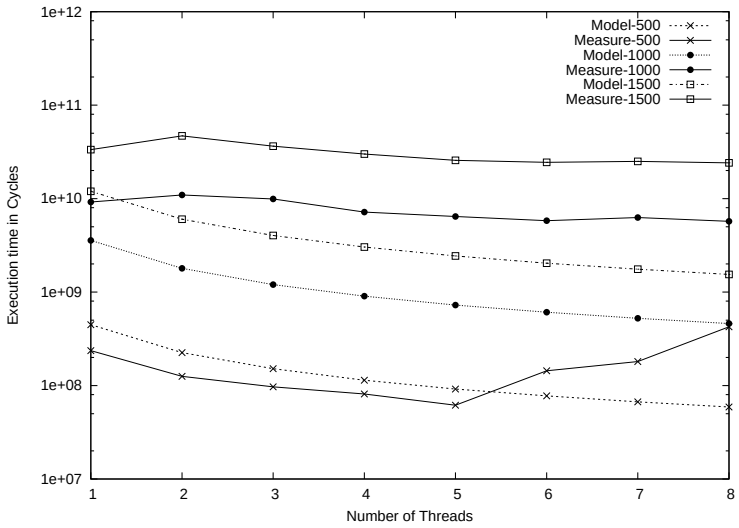

Figure 5.11: Modeling vs. measuring for MMM

To closely examine the model's absolute accuracy, we calculated the difference ratios between the modeled and measured results $((Modeled_c - Measured_c)/Measured_c)$. In this case, positive values imply overestimation while negative values mean underestimation. As illustrated in Figure 5.13, our current implementation leaves plenty of room for tuning and improvements for its absolute accuracy. Many factors may contribute to the inaccuracy, such as conducting a compile-time cost modeling relying on static analysis, an input high-level IR which is quite different from the final assembly code, as well as some ignored sources of execution cost including remote memory accesses, instruction cache misses, coherence cache misses, and bus/memory contentions.

The results of validating the major sub-models are given in Table 5.3 and Table

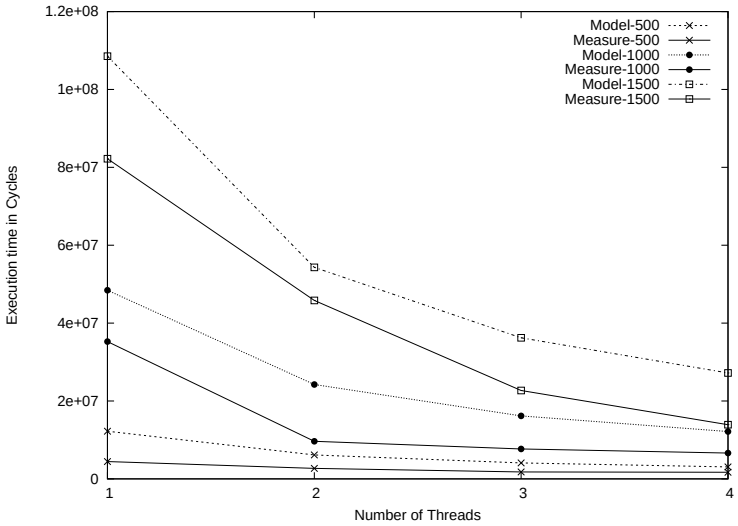

Figure 5.12: Modeling vs. measuring for Jacobi

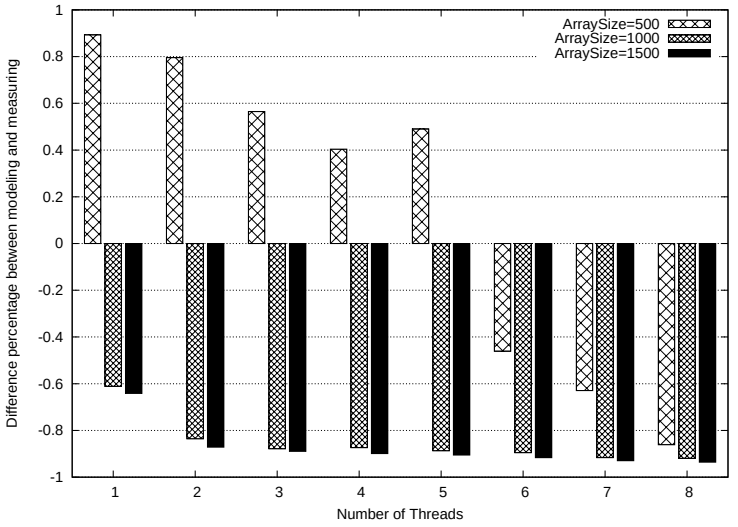

Figure 5.13: Absolute accuracy of modeling

5.4. Each table compares the categories of the measured cycles to the modeled results in terms of cycle count, percentage, and cycles per iteration. It is clear that the selected three categories (Unstalled cycles, FLP units, and D-cache stalls) in our mapping rules contributed the most ($> 99\%$) to the final measured cycles. Comparing D-cache stalls to the modeled cache cost reveals that the cache model is the major culprit for the inaccuracy of our model. For MMM-1000, the modeled cache cost is only 0.036 cycles/iteration while the D-cache stalls account for 6.95 cycles/iteration in the measured results. Similarly, the modeled cache cost for Jacobi-1000 is 35.92/iteration while the corresponding measured D-cache stalls is only 2.49 cycles/iteration. Accurate memory access patterns such as stack distances [32] might have to be considered to get better cache cost estimations in addition to the total amount of memory touched by a program. The processor model also needs further improvement regarding the floating point register dependence cost since it underestimates FLP units stall for Jacobi kernel.

The results of modeling OpenMP with `schedule` clause are given in Figure 5.14 using array size 1000×1000 with 4-thread execution. Only `static` scheduling results are shown because `dynamic` and `guided` scheduling have very similar results using our current implementation. Our OpenMP cost model is promising in that it is capable of capturing the relative performance of different chunk sizes of static scheduling, though the absolute accuracy is not very satisfactory. The modeled result successfully mimics the curve of the measured results. As illustrated in Figure 5.15, the left end of the curve indicates excessive scheduling overhead when a small chunk size is used in the schedule, and the right end of the curve captures the load imbalance among

Measured results			
Category	**Cycles**	**Percent**	**Cycles/Iter**
Unstalled cycles	9.95E+8	9.00%	0.995
FLP units	3.1E+9	28.04%	3.10
D-cache stalls	6.95E+9	62.87%	6.95
Branch misprediction	2.89E+6	0.026%	2.89E-3
Instruction miss	6.24E+6	0.057%	6.24E-3
RSE stalls	2063.3	1.87E-05%	2.06E-06
GR scoreboarding	51.67	4.67E-07%	5.17E-08
FE flushes	2.38E+5	2.15E-03%	2.38E-4
Total	1.11E+10	100%	11.06
Modeled results			
Category	**Cycles**	**Percent**	**Cycles/Iter**
Processor	1.5E+9	27.1%	1.5
Loop	4.004E+9	72.3%	4.004
Cache	3.587E+7	0.65%	0.036
OpenMP	3000	4.92E-05%	3E-6
Total	5.54E+9	100%	5.54

Table 5.3: Breakdown of cycles for MMM-1000

Measured results			
Category	**Cycles**	**Percent**	**Cycles/Iter**
Unstalled cycles	1.01E+7	28.76%	10.11
FLP units	2.25E+7	64.12%	22.53
D-cache stalls	2.49E+6	7.09%	2.49
Branch misprediction	1228.82	3.5E-3%	1.23E-3
Instruction miss	7668.13	0.02%	7.67E-3
RSE stalls	7.72	2.2E-05%	7.72E-06
GR scoreboarding	0.31	8.82E-07%	3.1E-07
FE flushes	49.45	1.41E-04%	4.94E-5
Total	3.51E+7	100%	35.13
Modeled results			
Category	**Cycles**	**Percent**	**Cycles/Iter**
Processor	8.5E+6	17.55%	8.5
Loop	4.004E+6	8.27%	4.004
Cache	3.59E+7	74.17%	35.92
OpenMP	3000	5.62E-05%	2.72E-3
Total	4.84E+7	100%	48.43

Table 5.4: Breakdown of cycles for Jacobi-1000

thread when a large chunk is assigned in each dispatch. As a result, our model can help the compiler and runtime system to choose a proper choice of the chunk size considering the trade-off between scheduling overhead and load balancing.

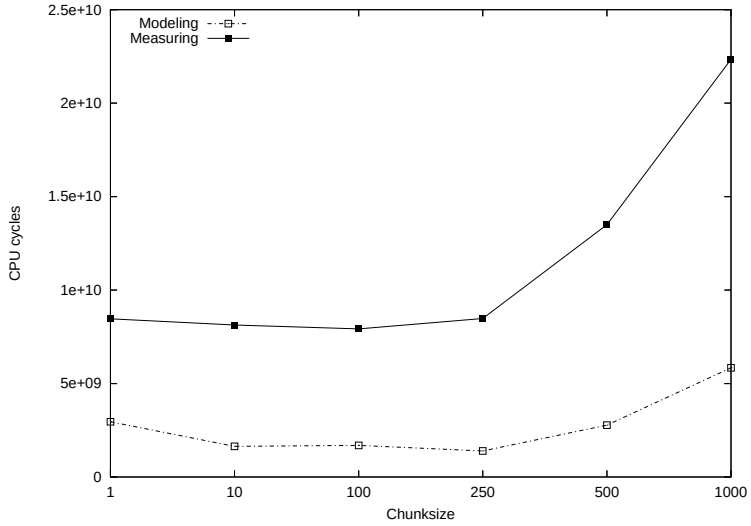

Figure 5.14: Modeling `schedule(static,n)`

5.5 Related Work

Building analytical models that incorporate knowledge of both hardware and software has been an attractive and challenging research topic since the early days of computing. We only mention a few of them below since an exhaustive list is beyond of the scope of this dissertation.

Numerous compiler cost models for computation and memory access have been proposed, especially for guiding loop transformations [113]. Wang [110] presents

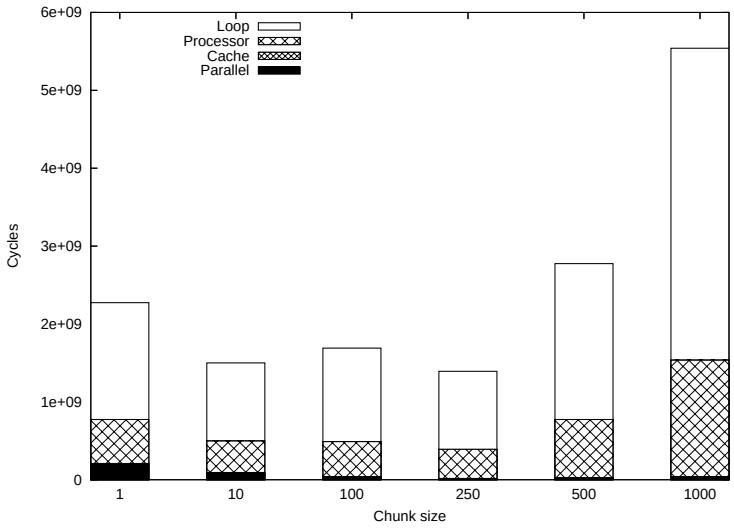

Figure 5.15: Breakdown of modeled cycles for `schedule(static,n)`

a cost model for superscalar processors considering multiple function units and instruction dependencies. His work maps high level language constructs to low level machine operations using cost tables for each basic block. Symbolic aggregation of cost for the basic blocks are used for unknown control flow information. Porterfield [91] defines the overflow iteration as the maximum number of iterations of a loop that have all the data access without any cache miss. Ferrante *et al.* [34] determines the innermost loop with overflowing caches using the number of distinct cache lines accessed inside a loop to guide transformations like loop interchange. McKinley's cache model [76] is based on equivalence classes of array references showing temporal and spatial locality. Ghosh *et al.* [40] introduces cache miss equations based on a system of linear Diophantine equations from a reuse vector. A more recent study by Yotovo *et al.* [115] demonstrates the use of a detailed compile-time model for the sequential matrix multiplication kernel and compares it with a library using empirical

optimizations.

It is common for models to rely on benchmarking and profiling. Saavedra *et al.* [94] uses benchmarking of abstract operations to find characteristics of both applications and machines, trying to estimate execution time for arbitrary machine/program combinations. Based on the control flow graph edge frequency and memory reuse distance obtained from binary profiling of benchmarks, Marin *et al.* [72] apply approximation functions to build parameterized performance models. Snavely *et al.* [98] use benchmarks to probe the machine's signature and use a profiling tool to generate an application profile. A convolution model based on machine and application features was applied to predict performance. However, their model only targets memory-intensive applications.

Models [10, 99, 52] are also used to estimate the performance of parallel programs. Some of them [30] are at a high level, proving qualitative insights. Some are based on task graphs [9]. Several integrated performance frameworks [8, 83] also exist. However, they are often not lightweight and versatile enough for compile-time usage. There are also several compile-time parallel models. A simple heuristic based on the size of the loop body and iteration space is used in SUIF [64] to decide if parallelization is profitable. A simple fork-join model [108] is used to derive a threshold for dynamic serialization of OpenMP. None of the existing models consider sufficient OpenMP details such as scheduling and chunk sizes for more serious usage. Considering coherence misses, lock time and memory contention, Jin *et al.* [49] presents an analytical model for SMPs to select optimizations for reductions, but not for OpenMP applications.

Chapter 6

Towards An OpenMP Cost Model for Multicore

In this chapter, we explore the impact of the multicore architectures on OpenMP cost models based on our evaluation of OpenMP on such platforms. Several possible extensions of our model are discussed to accommodate the multicore platforms.

6.1 Evaluating OpenMP on CMT

The chip multithreading technologies (introduced in Section 3.4) bring new opportunities and challenges for OpenMP. As the major player in expressing thread level parallelism, OpenMP is a promising candidate to exploit those new processors and the corresponding hierarchical SMPs consisting of CMT processors. However, OpenMP

may need to identify sibling processors[1] to perform the work cooperatively with proper scheduling and load balancing mechanisms. OpenMP has to avoid inter-thread competition for shared resources, and to select the best number of cores from a group of multiple cores. With those questions in mind, we designed several experiments to study the behavior of current OpenMP on multicore and multithreaded architectures [63].

6.1.1 Methodology

Two different systems were chosen to evaluate OpenMP on CMT platforms: a 4-way Sun Fire V490 with dual-core UltraSPARC IV processors and a 2-way Dell Precision 450 workstation with Xeon-HyperThreading processors. OpenMP's performance and overhead is studied using the EPCC Microbenchmark suite and a subset of the benchmarks in the NAS parallel benchmark 3.0 suites [48].

The Sun Fire V490 server (shown in Figure 6.1-(a)) for our experiments has four 1.05 GHz UltraSPARC IV processors (Shown in Figure 3.4-(a)) and 32 GB main memory. The basic building block is a dual CPU/Memory module with two UltraSPARC IV cores, an external L2 cache and a 16 GB interleaved main memory. The Sun Fireplane Interconnect is used to connect processors to memory and I/O devices. It is based on a 4-port crossbar switch with a 288-bit (256-bit data, 32-bit Error-Correcting Code) bus using a clock rate of 150 MHz. The maximum transfer rate is thus 4.8 GB/sec. The Sun Fire V490 is loaded with Solaris 10 operating system

[1]We use the term "sibling" to refer to the two cores in the same processor for CMP, and the two logical processors in the same physical processor for SMT.

and Sun Studio 10 integrated development environment which supports OpenMP 2.0 APIs for C/C++ and Fortran 95 programs. Solaris 10 allows superusers to enable or disable individual processor cores by using a processor administration tool named `psradm`. An environment variable `SUNW_OMP_PROBIND` is available for users to bind OpenMP threads to processors.

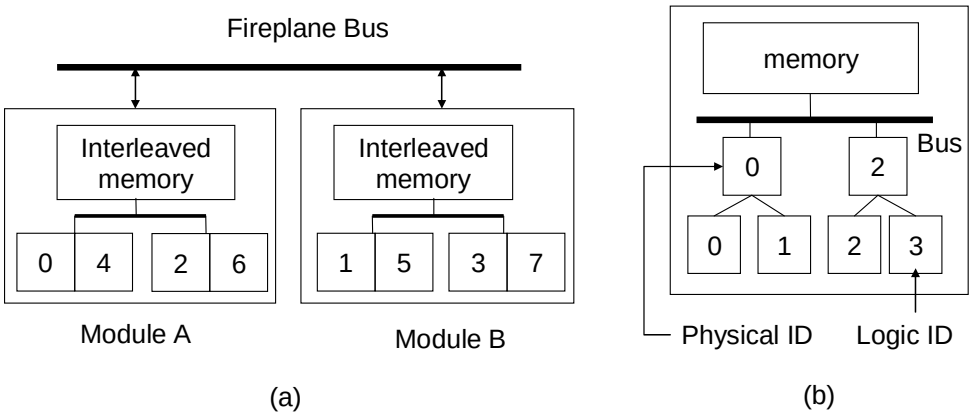

Figure 6.1: Dell Precision 450 and Sun Fire V490

The Dell Precision 450 workstation (shown in Figure 6.1-(b)), on which we carried out the experiments, has dual Xeon 2.4 GHZ CPUs with 512K L2 cache, 1.0GB memory and HyperThreading technology (Shown in Figure 3.4-(b)). The system runs Linux with kernel 2.6.3 SMP. The Omni compiler [95] is installed to support OpenMP applications and GCC 3.3.4 acts as a backend compiler. Linux Kernel 2.6.3 SMP in our 2-way Dell Precision workstation has a scheduler which is aware of HyperThreading. This scheduler can recognize that two logical processors belong to the same physical processor, thus maintaining load balancing per physical CPU, not per logical CPU. We confirmed this via simple experiments that showed that two threads were always allocated to two different physical processors unless there was a

third thread or process involved.

Since it was not clear whether the Solaris 10 on Sun Fire V490 with 4 dual-core processors is aware of the asymmetry among underlying logical processors, we used the Sun Performance Analyzer [104] to profile a simple OpenMP Jacobi code (Shown in Figure 5.10(b)) running with 2, 3, and 4 threads. We set the result timelines to display data per CPU in Sun Performance Analyzer, and found that Solaris 10 is indeed aware of the differences in processors of a multicore platform and tries to avoid scheduling threads to sibling cores. Therefore, we can roughly assume that the machine acts like a traditional SMP for OpenMP applications with only 4 or less active threads. For applications using 5 or more threads, there must be sibling cores working at the same time. As a result, any irregular performance change from 4 to 5 threads might be related to the deployment of sibling cores.

The EPCC Microbenchmark Suite [19] and NAS parallel benchmark (NPB) 3.0 [48] were chosen to discover the impact of CMT technology on OpenMP applications. The EPCC microbenchmark is a popular program to test the overhead of OpenMP directives on a specific machine while the NAS OpenMP benchmarks are used as representative codes to help us understand the likely performance of real OpenMP applications on SMP systems with CMT.

For experiments running on the Sun Fire V490 machine, we compiled the NAS parallel benchmark using the Sun Studio 10 compiler suite with the generic compilation option `-fast -xopenmp` and ran them from 1 to 8 threads in multi-user mode. After that, we compare the EPCC benchmark performance in 2 threads on two sibling cores and non-sibling cores (in fact, the former reflects the effects of CMP and

the latter the effects of the traditional SMP), and in 4 threads on four cores from two processors(a CMP configuration) and from four different processors (a traditional SMP configuration), respectively. In order to ensure the desired core/processor layout, we take advantage of the `prsadm` utility from Sun Solaris 10 to turn off the cores that we will not use.

The experiments on the Dell Precision workstation were designed in the same fashion whenever possible. For example, we measured the performance of the EPCC microbenchmarks in 2 threads using 2 logical processors of the same physical processor (a SMT configuration) and on 2 physical processors with HyperThreading disabled (an approximate traditional SMP). This way, we can understand the influence of HyperThreading better.

6.1.2 Results

This section presents the results of the experiments on the two machines using the selected benchmarks, and our analysis.

Figure 6.2 shows the OpenMP synchronization overhead ratio for OpenMP directives on cores belonging to the same processor(s) and different processor(s). OpenMP directives on CMP take slightly less time than on a traditional SMP except for three mutual exclusion directives: `CRITICAL`, `LOCK` and `ATOMIC`. The overall overhead difference tends to be smaller (close to ratio of 1) when more threads are used, except for the `ATOMIC` directive. Therefore, we may conclude that sibling cores do not bring significantly faster synchronization or fewer overhead for OpenMP constructs on the

Sun Fire V490 machine. This is because the sibling cores in the UltraSPARC IV do not share L1 and/or L2 cache to facilitate faster communications. In contrast, we display the overhead ratio between the two sibling logical processors and two physical processors of Xeon-HT in Figure 6.3. Only the overhead of `ATOMIC` is smaller in the case of sibling processors. The reason is that OpenMP synchronization directives are mostly implemented using spin-wait in current OpenMP implementations, which leads to more competition for the shared resources on the Xeon-HT system.

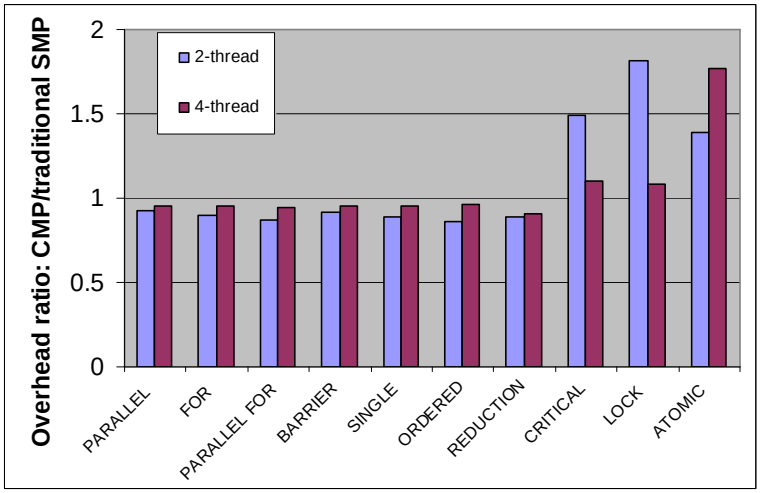

Figure 6.2: Synchronization overhead ratio: CMP vs. traditional SMP

The execution of the NAS Parallel Benchmark (NPB) [48] confirmed acceptable performance on SMPs with CMP processors as shown in Figure. 6.4 (although this may be compromised when the threads compete for bandwidth to memory, as shown from 4 to 5 threads where 2 threads out of 5 are running within two cores on one processor), but not on SMPs with SMT CPUs depicted in Figure 6.5. The best performance is not always achieved using the maximum number of available threads on SMT processors: the typical similarity of work carried out by OpenMP threads in

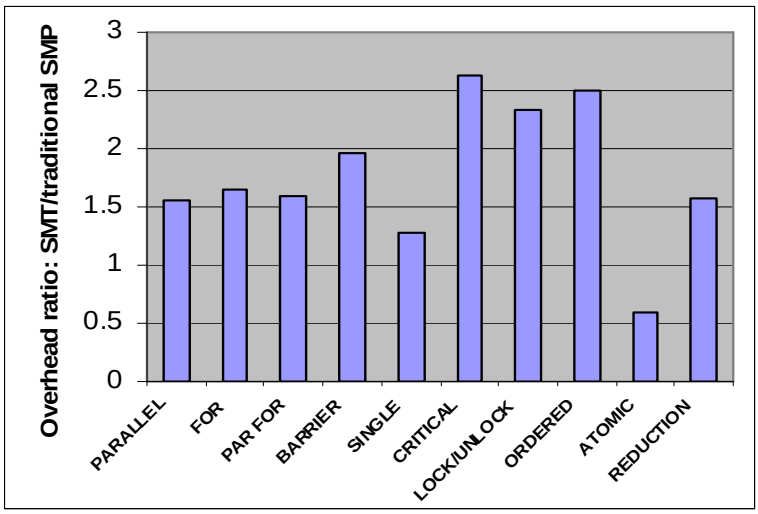

Figure 6.3: Synchronization overhead ratio: SMT vs. traditional SMP

a team, and the default static scheduling policy, tend to stress the shared resources in SMT processors and thus often lead to performance degradation. In conclusion, existing OpenMP compilation and execution strategies are not able to make the most out of new HSMPs or even worse: they may lead to performance degradation.

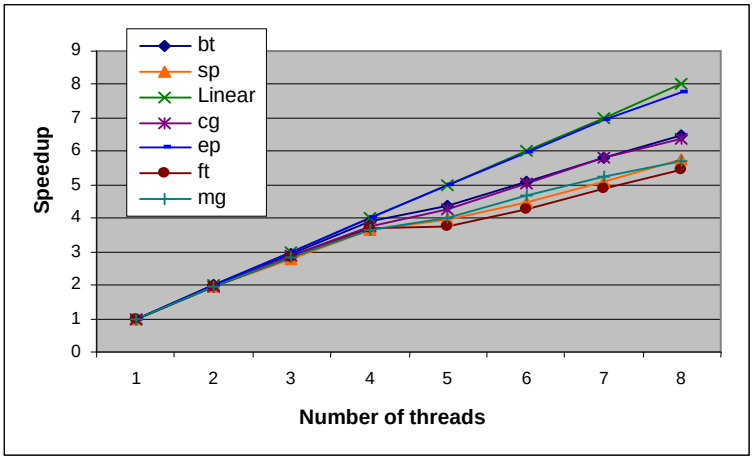

Figure 6.4: Speedup of NPB on CMP

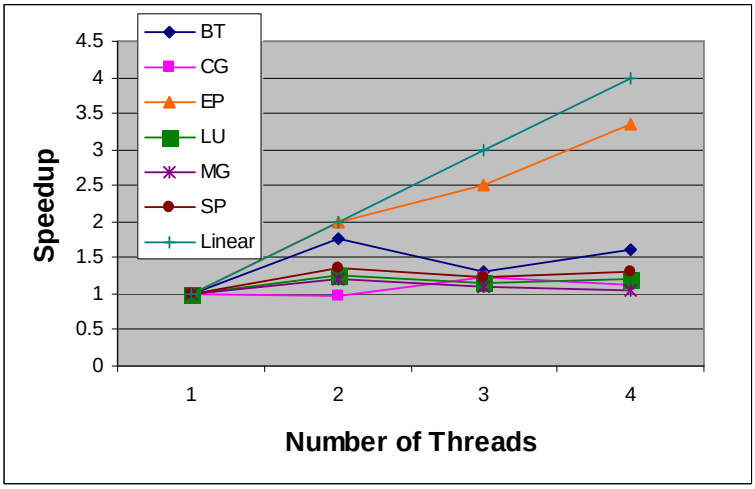

Figure 6.5: Speedup of NPB on SMT

6.1.3 Conclusion

The Sun Fire V490 is a successful platform for OpenMP as one of the first generation CMP (multicore) machines. We find its overall scalability to be comparable to a traditional SMP machine since each core has the similar capability as a regular uniprocessor. Most OpenMP applications from NPB 3.0 scale very well. For memory-intensive applications using 5 and more threads, scalability may be compromised to an acceptable degree due to the competition for the shared off-chip data path between sibling cores. There should be a threshold for the applications' memory demand to be intensive enough to cause performance degradation on a specific machine. Unfortunately, the EPCC microbenchmark's results did not show profitable faster synchronization among sibling cores within one processor in this machine. We believe it is mostly due to the fact that the L2 cache is not really shared between sibling cores. Meanwhile, the Solaris operating system schedules OpenMP threads very well

by taking the asymmetry between cores into consideration.

Compiler optimizations are also key factors in OpenMP performance on the Sun platforms. For example, we observed a performance degradation from 4 to 5 threads for Jacobi code compiled with `-fast -xopenmp` even for a small data set (3 500x500 arrays and 1000 iterations). The Analyzer showed that the Sun Studio 10 compiler performed loop unrolling on the major loops and inserted `PREFETCH` instructions into them. Data cache stall information collected from hardware counters hints that some `PREFETCH` operations initiated from two sibling cores stressed the shared data path connected to L2 cache and memory, thus incurred longer stall cycles than usual. But it is not always the case for all `PREFETCH` operations. Further work is needed to understand the exact conditions and effects of these traditional optimizations with regard to the OpenMP performance on machines with CMP capabilities.

On the other hand, we observed that a straightforward OpenMP implementation for traditional SMP architectures may not achieve good scalability on the Xeon-HyperThreading system. The main reasons are the memory bandwidth bottleneck [41] and the competition for the shared computing resources, which degrade the overall SMT performance. Adaptive OpenMP scheduling policies [116] and precomputation/prefetch approaches via a helper thread [111] might be able to lead to a better performance. Our experiments also showed that an SMT-aware OS is important for maintaining load balance and efficiently utilizing the resources of an SMT system. Finally, the EPCC microbenchmarks results indicated that the overhead of OpenMP synchronization implementation in a SMT system is higher than that in a SMP system. Thus, the current OpenMP implementation needs to take the SMT

features into account.

6.1.4 Related Work

Other research evaluating OpenMP on multi-core processors [11, 31] have similar findings. Many researchers have investigated the underlying reasons and tried new ways to addresses the new challenges. Frumkin [37] demonstrated that data locality of the shared L2 cache is important for scalability on a multi-core POWER4 system. Lee *et al.* [60] showed the negative effect of the shared data path for memory-intensive applications on SUN UltraSPARC IV dual-core systems.

SMT technology has been more extensively studied during the past decade. Various application experiences [66, 70, 92] on SMTs have reported that a complementary workload is less likely to introduce resource conflicts among threads and fits best for SMT processors. Many strategies [15, 82, 75] have been developed for efficient sharing of key resources, especially cache. Helper threads have been proposed to prefetch critical data for a thread in an SMT processor in numerous studies [67, 111, 54, 109]. Zhang *et al.* [116] demonstrated an empirical adaptive scheduler and the improved performance on SMT processors. Heterogeneous multi-core processors have been explored for application performance [16] and compiler optimizations [33]

Most existing research [60, 11, 111] has not studied the interaction of CMP and SMT technology. Further, most studies have focused on one aspect of OpenMP execution, e.g. the scheduling policy [116] or the number of threads [31].

6.2 Impact on OpenMP and Its Cost Models

With the popularity of multicore processors and their diverse designs, OpenMP faces numerous opportunities and challenges. Consequently, OpenMP cost models will play even more important roles in many different scenarios and building an accurate and efficient OpenMP cost model will become more difficult than before.

Multicore architectures will dramatically broaden the use of OpenMP for high-end computing, desktop applications, as well as embedded systems. An immediate result of the multicore trend is that systems with a wide variety of chip level multithreading capabilities will be widely deployed in the future and the number of threads that may be used to execute a program may be much larger than that supported by current SMPs. OpenMP is expected to be one of the major candidates for exploiting the abundant thread level parallelism in the multicore platforms, thus, many more applications will be written in OpenMP. In the meantime, as more SMP clusters uses multicore processors in the high performance computing market today, it will become a standard to use OpenMP with MPI to provide fine-grain parallelism and to improve load balancing in MPI code and to get addition speedup.

On the other hand, OpenMP has to be versatile to exploit the diverse multicore architectures, which differ from traditional multiprocessors using symmetric multiprocessing in many aspects. First, multicore architectures provide very high bandwidth among cores within the same chip with ultra low communication latency.

Second, chip-level integration of multiple processing units stresses the off-chip bandwidth and this may lead to performance degradation as demonstrated in our evaluation. Third, multicore architectures also greatly differ from each other in terms of the nature of hardware resource sharing, inter-core connections and supported logical threads per core. How to map the applications to hardware resources and whether or not to use all available threads becomes a tricky question depending on hardware profiles and application features. Finally, future multicore architectures tend to have heterogeneous cores to efficiently meet different computation, memory, communication requirements from user applications. Knowledge about hardware capability and layout as well as application features is needed to assign the right type of work to the right kind of cores during the compilation phase or at runtime. OpenMP language and implementations have to consider all these unique hardware features to fully take advantage of multicore architectures while avoiding their limitations.

As one of the major motivations behind multicore architectures, increasing performance while keeping energy consumption under control should also become a major goal for OpenMP. Almost all current multicore processor designs incorporates energy saving features to enhance performance-per-watt via dynamic voltage and frequency scaling(DVFS). Intel even introduces Ultra Fine Grained Power Control over Core 2 Duo processors to turn off unused function units inside a processor core. Research in power-aware computation is a very active area today: For a fixed power budget, Annavaram *et al.* [13] increases the frequency of one processor core and deactivates others to fasten the execution of sequential parts in an effort to mitigate the limits from Amdahl's law. DVFS is widely used to tune down the processors running idle

MPI communication and collective operations [53, 39]. Similarly, in shared-memory programming such as OpenMP, some researchers [61, 65] have exploited the idle time caused by imbalanced barriers. As a result, OpenMP has to have some inherent support to enhance its energy-efficiency to meet this new requirement.

However, current OpenMP implementations [62, 95, 18, 106, 104] utilize some fixed, predefined approaches to transform and execute OpenMP programs, regardless software and hardware varieties, as well as energy concerns. For example, the loops with OpenMP directives are translated into outlined functions (or inlined microtasks) in compilers; OpenMP applications are executed using a set of default parameters, like using maximum available number of processors as the number of threads, statically dividing loop iteration space into equally chunks and assigning them to the threads, randomly mapping threads to underlying processors and so on. The simplified approaches are primarily designed for flat, uniform access shared-memory space (UMA) systems and for relatively modest thread counts. They usually work well for applications with regular loop-based, computation-intensive workload on conventional small-scale SMPs. But they may lead to performance degradation for CMT platforms as shown in our evaluation in the previous section. As to energy-efficient computing,it is still largely ignored in current OpenMP compilers and runtime systems.

One of the major approaches to addressing the challenges posed by diverse applications and new platforms is to use adaptive runtime systems with empirical search. This works in the following two steps: one is to try different execution configurations for a piece of code; the other is to choose the best one among all tried variants based

on their measured performance. For example, Zhang *et al.* [116] proposed an empirical runtime search for choosing the right number of threads and a good scheduling method for iterative applications on SMPs with HyperThreading. While empirical search usually works well for iterative algorithms and a small search space consisting of limited factors, it fails to handle more complex situations such as non-iterative algorithms and a much larger search space considering more variants including different chunk sizes and thread-processor(core) bindings.

We believe that extending current OpenMP cost models, including ours, for multicore architectures can help to address the challenges by providing accurate estimation of execution time and power consumption of OpenMP applications on such platforms. In particular, OpenMP implementations built on top of cost models can provide alternative and complementary ways to enhance the application- and platform-awareness of OpenMP compilation and runtime support in an energy-efficient way, especially for non-iterative algorithms and for a large search space.

Figure 6.6 illustrates a high level overview of a model-driven OpenMP implementation for multicore architectures. The left part of the figure shows an extended OpenMP cost model with enhanced hardware sub-models to reflect new multicore features: The processor model should be adapted to describe recent multicore processors such as instruction set associated with both cycle and energy cost, numbers of available function units, latencies, and registers. The cache model will be used to predict both cache usage and the associated energy consumption. New penalties should be introduced to consider cache contention delay. Finally, the topology

model will group processors and caches together to form a SMP environment considering their interconnect. The resulting OpenMP cost model is expected to detect inter-thread resource contention and output metrics considering both performance improvement and energy consumption.

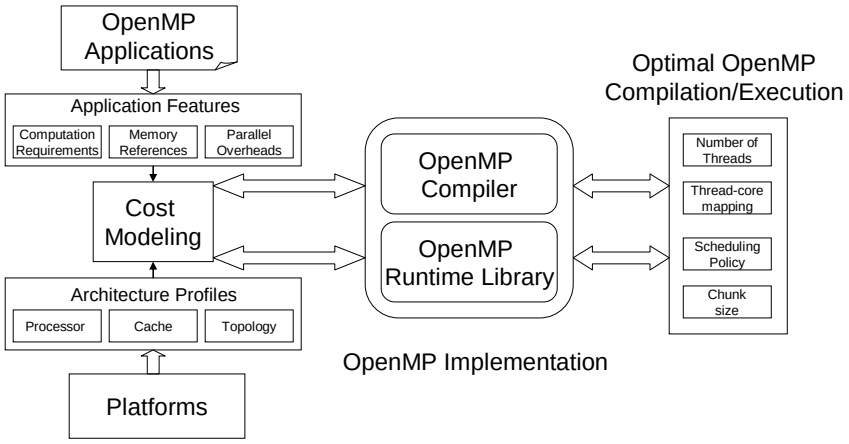

Figure 6.6: Model-driven OpenMP implementation for multicore

Based on the modeling results, compilers and runtime libraries can guide transformations and suggest execution parameters. Possible usage of the model-driven OpenMP implementation includes the following: the analysis and prediction of differently-behaving threads (e.g. memory or I/O intensive), giving advices on whether SMT threads should be used, and optimal OpenMP parameters (the number of threads for a construct, task-thread-core mapping, loop scheduling policy, and loop chunk size for each thread).

In the following section, we discuss some details about the model extensions.

6.3 Extensions of OpenMP Cost Models for Multicore

We believe that OpenMP cost models for multicore architectures could be widely deployed in the near future to enhance the platform-awareness of OpenMP implementations and to improve the energy-efficiency for both pure OpenMP and hybrid MPI/OpenMP applications. We explore several major extensions of our OpenMP cost model for multicore architectures in this section.

6.3.1 Modeling Contentions

The contention model for multicore platforms should consider various new sources, including off-chip data path contention and inter-thread cache contention, for contention delays among threads. The major difficulties of contention modeling are the non-deterministic behavior of threads during execution and accurate prediction of resource requests from each thread. Some statistical input such as average request rate from threads and average service time of the resource are often used to simplify the problem.

The contention cost for off-chip data path can be usually modeled using a queuing model [55] like the traditional models [49] for bus and memory contentions. We can view each resource shared by multiple threads as an independent FIFO queue serving multiple clients, denoted as $M/D/n$ using Kendall's notation from queuing theory. In this notation, M indicates a Markovian (exponential) distribution for inter-arrival

times of clients, which is the characteristic of a random thread. D implies that the service time for the resource to finish the request from a thread is deterministic and constant. n means there are n service channels in the modeled resource. The value of n should be 1 if the modeled resource can only serve one request at a time. The queuing model provides a set of standard formulas for various performance metrics, including average number of clients waiting in the queue, average waiting time in the queue, the probability of the queue is full, and so on. Contention modeling is mostly related to the average waiting time in the queue, which can be represented using the following equations:

$$\begin{aligned} T\_contention_c &= (U * S)/(2 * (1 - U)) \\ U &= (t * R)/(1/S) \end{aligned}$$

In the equations above, S is the average service time for each request and U is the utilization rate of the modeled resource by all threads, which in turn can be calculated from the number of threads t, the average resource request rate R per thread, and service rate $1/S$.

Modeling inter-thread cache contention caused by competing threads needs the knowledge of thread-core mapping information (we will discuss it in Section 6.3.2) and accurate cache access patterns per thread [43] for all threads mapped to the cores connected to a shared cache.

To model cache contention, we could borrow some ideas from Chandra `et al.`'s inductive probability model [23]. Their model's input are sequence profiles and circular sequence profiles of participating threads, denoted as $seq_x(d_x, n_x)$ or $cseq_x(d_x, n_x)$

if the modeled thread is called X. The sequence profile is a sequence of n_x memory accesses to d_x distinct addresses by a thread mapped the same cache set. A circular sequence profile is a special sequence profile: its first and the last accesses are to the same address. For example, (A, B, A) is a circular sequence. Predicting cache misses for a single thread is easy if its circular sequence is available: the last access is a cache miss if and only if $dx > CacheAssociativity$. For two-thread programs without shared data, the cache miss condition is that during the lifetime of $seq_x(d_x, n_x)$, if there is an intervening accesses by thread Y, denoted as $seq_y(d_y, n_y)$, the last access of thread X is a miss if and only if $d_x + d_y > CacheAssociativity$. Using inductive probability formulas considering all possible interleaved access ways making the cache miss condition hold, their model tries to estimate the extra cache misses caused by the intervening thread Y to thread X. Although their model only considers two threads without any shared data, it is possible to extend it to handle more threads considering shared data.

6.3.2 Modeling Topology

Topology models become very necessary given several prominent asymmetric features in multicore architectures. As shown in Figure 3.4, SMPs containing several multicore processors with SMT technology forms a hierarchical structure. The computation capacities of different threads are quite different from one depending on a multiplexed logical processor to another using a dedicated physical processor. Heterogeneous multicore processors make the computation asymmetry even more obvious. In addition, the interconnections between processors or cores are also more complex

than those in symmetric multiprocessing. There is a huge difference between chip-level connection and inter-chip connection in terms of bandwidth and latency. As a result, an OpenMP application running on the asymmetric multicore platforms might behave quite differently when different thread-core mapping strategies are used.

The topology extension of our OpenMP cost model for multicore architectures should reflect the asymmetry in the platforms in terms of computation capacity and communication bandwidth and latency. The whole system's topology can be modeled by a weighted graph, in which a node represents a processor or core and an edge corresponds to an interconnection. The weights of each edge specify the bandwidth and latency of the interconnect between two processors or two cores.

Various thread-core mapping strategies of OpenMP applications can be evaluated using the topology model. Mapping heuristics become easy to implement with the knowledge of both application features and system topology information. For example, threads with frequent synchronization operations could be allocated to cores connected with a fast connection. Threads with overlapped memory accesses should be mapped to cores with a shared cache to increase the resource utilization and minimize cache misses. Complimentary workloads such as memory-intensive thread and I/O-bound thread can be scheduled to multiplexed logical processors. For heterogeneous multicore platforms, threads with different computation requirements will be mapped to the right cores to achieve optimal resource utilization with high performance.

6.3.3 Modeling Energy

One of the major extensions of our OpenMP cost model for multicore architectures will be an energy sub-model. This extension involves modifications to almost all major hardware component models including those for processor, memory hierarchy and interconnect because power consumption happens in each of them. The total energy consumption for executing OpenMP code is built on top of several low-level energy models for computation, memory access, and interconnect energy usage.

The basic idea for adding energy modeling into those hardware models is to associate each basic operation with a certain amount of power consumption. For example, the processor model will include quantified power consumption per instruction. Similarly, each memory access to a level of memory hierarchy could also have a certain amount of energy consumption. The core of the energy model is a power cost table that records energy cost for each type of instruction and memory access. The values for the energy cost could be obtained by directly measuring the power consumption using some microbenchmarks containing a set of processor instructions or memory accesses. One possible way to directly measure the power consumption for applications could be based on hardware counters as proposed by Isci *et al.* [45]. The average power consumption per instruction or per memory access is treated as the cost value in the table. More accurate energy modeling should consider inter-instruction effects and memory locality among memory accesses, with higher complexity.

With the additional energy model, the estimated cost will not solely depend on machine cycles, but will also consider the consumed power. A balanced consideration

of both performance and energy such as the $Energy * Delay^2$ metric [74] could be used as a final guideline to compare the energy-aware performance among different transformation or execution variants.

6.4 Summary

Extending an OpenMP cost model for traditional SMP platforms to the SMPs using multicore processors requires the consideration of several interrelated factors. Most of them come from the unique hardware features found in the multicore architectures such as the inter-thread cache contention and the hierarchical topology. Energy modeling is another major factor given the demanding requirement for efficient computing in an economical and ecological way. We have only given some initial explorations due to the resource limitations in this dissertation.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this dissertation, we have presented our experience of building a portable and optimizing OpenMP compiler based on Open64, which has motivated and enabled us to build an OpenMP cost model considering language details along with application features and hardware profiles. The resulting model is aimed to be accurate and efficient to be widely used in different scenarios, including guiding OpenMP compilation, helping runtime support, and enhancing hybrid MPI/OpenMP execution. Moreover, the emerging multicore, multithreaded architectures have been evaluated using OpenMP applications to explore possible extensions to model those new platforms.

In summary, this dissertation has the following contributions:

- an OpenMP reference compiler using a hybrid design which combines both native compilation and source-to-source translation approaches to achieve both portability and optimization in a single compiler.

- an implementation and evaluations of our design based on a real world, commercial-quality compiler, Open64, as well as its major branches. We ported the front-tends from Pathscale 2.1 to Itanium and merged them into OpenUH to enable complete support for OpenMP parsing. The original OpenMP transformations in Open64 and a runtime library based on ORC-OpenMP patch were enhanced to support more OpenMP constructs including `threadprivate`, `copyprivate`, and `copyin`. The `whirl2c` and `whirl2f` toolset has been augmented to emit portable and compilable code from Mid Level WHIRL in addition to High Level WHIRL, with a lot of bug fixes. As a result, OpenUH facilitates a wider range of OpenMP-related research than other research compilers.

- a novel OpenMP cost model has been proposed. To our best knowledge, our model is the first one considering enough OpenMP language details including overheads from major OpenMP constructs and load balancing among different threads. Moreover, a prototype has been implemented using the OpenUH compiler by adapting its cost models of sequential loops to model OpenMP loops. Our evaluation is also one of the very few attempts to measure the accuracy of both absolute and relative accuracy for a compile-time cost model.

- one of the pioneering, detailed evaluations of OpenMP on multicore and multi-threaded platforms exposing the new opportunities and challenges for OpenMP

and its cost models. We further explore some potential extensions to OpenMP cost models for the new platforms, including sub-models for cache contention, topology, and energy consumption.

In conclusion, we have found that Open64 is a very good open source compiler to conduct advanced development and research tasks. Its major advantages include multiple frontends for mainstream programming languages, abundance and stable analysis and optimization phases, modularized and extensible implementation, rich support for debugging and internal tracing, as well as numerous contributions from the user community. Consequently, as a branch of Open64, OpenUH inherits all the advantages of Open64 mentioned above along with a more complete and stabler support for OpenMP. OpenUH also significantly enhances its portability by improving the quality of the `whirl2c` and `whirl2f` toolset. The final performance of OpenUH is comparable with commercial OpenMP compilers and outperforms most source-to-source research compilers. The major downside of Open64 and OpenUH is that the learning curve for new compiler developers might be very steep due to the complexity of the compiler and the lack of good documentations.

As to building OpenMP cost models, the major difficulty is the deep knowledge needed about hardware, software and their internal and external interactions. Implementation is often very complicated given the numerous factors to be modeled. Our experience of building a portable and optimizing OpenMP compiler has provided us the right insight into the performance details of OpenMP such as scheduling policies and load balancing. The availability of cost modeling for sequential loops in Open64 has dramatically reduced many fundamental but tedious implementation tasks such

as building the IR-to-resource cost table and calculating memory footprints for applications. The resulting model can provide relatively accurate estimations for OpenMP applications directly from their source code.

However, some limitations do exist in our model. Like any other compile-time models, our model has a limited resolution when the values of some input parameters are unknown. A typical example is that the iteration count has to be estimated as a fixed default number like 100 when it is unknown during the compilation phase. In addition, the model works at a high level IR and is unable to predict the final form of the code after some complex, implementation-dependent optimizations conducted on low level IRs in the later phases, making it inaccurate. Third, interactions among different hardware/software components are largely simplified in our model: we consider one component at a time and combine their results in simple ways. The reason is that it would be prohibitively costly and sometimes impossible to consider all the correlations in the modern out-of-order, superscalar processors with multiple levels of non-blocking memory hierarchy. For example, how to accurately convert cache misses into final execution latencies is still an open question due to the overlapping between computation and memory requests.

There are two other concerns with regard to a cost model in general. One is the trade-off between accuracy and efficiency. Theoretically, one can always try to improve the accuracy of the model by considering more and more factors. But the cost/accuracy ratio might not justify the efforts, especially for compile-time models which are invoked hundreds of times or more for one compilation unit. The other concern is the model's portability and reusability. There are still very few ways to

express the knowledge used to build models and to share the code of their components, yet this could save a lot of repetitive efforts to build various types of machine profiles/application signatures and to implement popular modeling components. Our current implementation has not considered the two factors and is thus not flexible and reusable enough.

Multicore and multithreaded platforms add several levels of difficulties to model OpenMP applications. They demand reevaluating and updating traditional models for processors, caches, buses and networking. The conventional notion of a fixed number of available resources does not directly apply to the new dynamic resource features from simultaneous multithreading in multicore processors. A faithful OpenMP cost model has to consider additional factors like the activated thread context and thread-processor mapping and bindings. Also, new performance factors have to be modeled to reflect more facts in the hardware and software interactions in multicore platforms, especially for cache sharing effect and energy efficiency. There are exponentially increased execution possibilities given the combinations of all the significant performance factors in multicore platforms.

Therefore, we believe that the value of cost models is greatest when they are used together with other approaches such as simulation-based and profiling/measurement-based performance prediction. A combined method can leverage advantages from different approaches and offer a good balance of accuracy, efficiency, portability and flexibility in performance prediction, especially for multicore platforms.

7.2 Future Work

In the future, we would like to continue our work on OpenUH to provide a versatile and user-friendly compiler infrastructure for OpenMP research and other research topics in general. In particular, we are interested in the following work:

- Conducting performance tuning on both the OpenMP translation and the runtime library.
- Completing the support for nested parallelism.
- Exploring and enhancing new OpenMP implementation techniques such as taskq [103] and adaptive runtime support [116].
- Studying the interactions between conventional sequential loop optimizations and OpenMP transformations.
- Developing more documents for developers.

The future work for the proposed OpenMP cost model can follow several directions and they are listed below:

- Adding extensions of the cost model to represent features for multicore platforms, including adding models for inter-thread cache contention, topology, as well as energy consumption.
- Parameterizing all the model results. For example, symbolic expressions [93] will be investigated to make the model results reusable by both static and

dynamic compilation components, as well as runtime libraries. This will also enable the cost model to support its combinations with simulation, profiling, measurement and runtime monitoring for enhanced accuracy.

- Exploring portable ways to build cost models is another major task. Instead of manually presetting performance parameters for target platforms, micro-benchmarking [114] will be added to obtain multiple levels of machine and application characteristics, ranging from cycles of basic operations to thread startup overhead, in a portable way.

- Finally, we will try to use our OpenMP cost model to provide efficient OpenMP translations and improve the quality of runtime support for various shared-memory platforms including multicore machines.

Bibliography

[1] "Intel itanium processor manuals," http://www.intel.com/design/itanium/manuals.htm.

[2] *APRIL: a processor architecture for multiprocessing.* New York, NY, USA: ACM Press, 1990.

[3] "AMD multi-core: Introducing x86 multi-core technology and dual-coreprocessors from AMD," http://multicore.amd.com/, 2005. [Online]. Available: http://multicore.amd.com/

[4] "GOMP - an OpenMP implementation for GCC," http://gcc.gnu.org/projects/gomp, 2005.

[5] "Kylin C compiler," http://www.capsl.udel.edu/kylin/, 2006.

[6] "The Open64 compiler," http://www.open64.net, 2006.

[7] "Top 500 supercomputers," http://www.top500.org/lists/2006/11, 2006. [Online]. Available: http://www.top500.org

[8] V. S. Adve, R. Bagrodia, J. C. Browne, E. D. A. Dube, E. N. Houstis, J. R. Rice, RizosSakellariou, D. J. Sundaram-Stukel, and P. J. T. M. K. Vernon, "Poems: End-to-end performance design of large parallel adaptive computationalsystems," *IEEE Trans. Softw. Eng.*, vol. 26, no. 11, pp. 1027–1048, 2000.

[9] V. S. Adve and M. K. Vernon, "Parallel program performance prediction using deterministic taskgraph analysis," *ACM Trans. Comput. Syst.*, vol. 22, no. 1, pp. 94–136, 2004.

[10] D. H. Albonesi and I. Koren, "An analytical model of high performance superscalar-based multiprocessors," in *PACT '95: Proceedings of the IFIP*

WG10.3 working conference on Parallelarchitectures and compilation techniques. Manchester, UK, UK: IFIP Working Group on Algol, 1995, pp. 194–203.

[11] G. S. Almasi, E. Ayguadé, C. Cascaval, J. Castaños, J. Labarta, F. M. X. Martorell, and J. E. Moreira, "Evaluation of OpenMP for the Cyclops multithreaded architecture," in *WOMPAT*, vol. 2716, June 2003, pp. 69–83.

[12] R. Alverson, D. Callahan, D. Cummings, BrianKoblenz, A. Porterfield, and B. Smith, "The Tera computer system," in *ICS '90: Proceedings of the 4th international conference on Supercomputing*. New York, NY, USA: ACM Press, 1990, pp. 1–6.

[13] M. Annavaram, E. Grochowski, and J. Shen, "Mitigating amdahl's law through epi throttling," in *ISCA '05: Proceedings of the 32nd Annual International Symposiumon Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 298–309.

[14] E. Ayguadé, M. Gonzàlez, X. Martorell, J. O. andJ. Labarta, and N. Navarro, "NANOSCompiler: A research platform for OpenMP extensions," in *the First EuropeanWorkshop on OpenMP*, Lund, Sweden, October 1999, pp. 27–31.

[15] F. Baboescu and D. M. Tullsen, "Memory subsystem design for multithreaded processors," UCSD, Tech. Rep., 1997.

[16] S. Balakrishnan, R. Rajwar, M. Upton, and KonradLai, "The impact of performance asymmetry in emerging multicore architectures," in *ISCA '05: Proceedings of the 32nd Annual International Symposiumon Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 506–517.

[17] J. Balart, A. Duran, M. Gonzalez, X. Martorell, and E. A. J. Labarta, "Nanos Mercurium: a research compiler for OpenMP," in *the 6th European Workshop on OpenMP (EWOMP'04)*, Stockholm, Sweden, October 2004.

[18] C. Brunschen and M. Brorsson, "OdinMP/CCp - a portable implementation of OpenMP for C," *Concurrency - Practice and Experience*, vol. 12, no. 12, pp. 1193–1203, 2000.

[19] J. M. Bull and D. O'Neill, "A microbenchmark suite for OpenMP 2.0," in *Proceedings of the Third European Workshop on OpenMP (EWOMP'01)*, Barcelona, Spain, September 2001.

[20] J. Bull, "Measuring synchronization and scheduling overheads in OpenMP," in *the Ruropean Workshop of OpenMP (EWOMP'99)*, Lund, Sweden, September 1999.

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

[22] W. W. Carlson, J. M. Draper, D. E. Culler, K. Y. E. Brooks, and K. Warren, "Introduction to UPC and language specification," Center for Computing Sciences, Tech. Rep., May 1999.

[23] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processorarchitecture," in *HPCA '05: Proceedings of the 11th International Symposium on High-PerformanceComputer Architecture.* Washington, DC, USA: IEEE Computer Society, 2005, pp. 340–351.

[24] B. Chapman, F. Bregier, A. Patil, and A. Prabhakar, "Achieving high performance under OpenMP on ccNUMA and softwaredistributed share memory systems," *Concurrency and Computation Practice and Experience*, vol. 14, pp. 1–17, 2002.

[25] B. M. Chapman, L. Huang, G. Jost, and H. J. B. R. de Supinski, "Support for flexibility and user control of worksharing in OpenMP," National Aeronautics and Space Administration, Tech. Rep. NAS-05-015, October 2005.

[26] W.-Y. Chen, "Building a source-to-source UPC-to-C translator," Master's thesis, University of California at Berkeley, 2005.

[27] Y. Chen, J. Li, S. Wang, and D. Wang, "ORC-OpenMP: An OpenMP compiler based on ORC," in *International Conference on Computational Science*, 2004, pp. 414–423.

[28] J. Corbalan, A. Duran, and J. Labarta, "Dynamic load balancing of mpi+openmp applications," *icpp*, vol. 00, pp. 195–202, 2004.

[29] I. corporation, "Intel multi-core processor architecture development backgrounder," http://www.intel.com/multi-core/docs.htm, 2005.

[30] D. Culler, R. Karp, D. Patterson, A. S. K. E. Schauser, E. Santos, and R. S. T. von Eicken, "Logp: towards a realistic model of parallel computation," in *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principlesand practice of parallel programming.* New York, NY, USA: ACM Press, 1993, pp. 1–12.

[31] M. Curtis-Maury, X. Ding, and C. D. A. D. S. Nikolopoulos, "An evaluation of OpenMP on current and emerging multithreaded/multicoreprocessors," in *FIRST INTERNATIONAL WORKSHOP on OpenMP*, Eugene, Oregon USA, June 2005.

[32] C. Ding and Y. Zhong, "Reuse distance analysis," Dept. of Computer Science, University of Rochester, Tech. Rep., 2001.

[33] A. E. Eichenberger, K. O'Brien, K. O. andPeng Wu, T. Chen, P. H. Oden, D. A. P. andJanice C. Shepherd, B. So, Z. Sura, A. W. andTao Zhang, P. Zhao, and M. Gschwind, "Optimizing compiler for the CELL processor," in *PACT '05: Proceedings of the 14th International Conference on ParallelArchitectures and Compilation Techniques (PACT'05).* Washington, DC, USA: IEEE Computer Society, 2005, pp. 161–172.

[34] J. Ferrante, V. Sarkar, and W. Thrash, "On estimating and enhancing cache effectiveness," in *Proceedings of the Fourth International Workshop on Languages andCompilers for Parallel Computing.* London, UK: Springer-Verlag, 1991, pp. 328–343.

[35] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, vol. C-21, pp. 948–960, 1972.

[36] M. P. I. Forum, "http://www.mpi-forum.org," 2007.

[37] M. Frumkin, "Efficiency and scalability of an explicit operator on an IBM POWER4system," NASA Ames Research Center, Tech. Rep. NAS-02-008, August 2002.

[38] "the GNU compiler collection," http://gcc.gnu.org, 2007.

[39] R. Ge, X. Feng, and K. W. Cameron, "Improvement of power-performance efficiency for high-end computing," in *IPDPS '05: Proceedings of the 19th IEEE International Parallel andDistributed Processing Symposium (IPDPS'05) - Workshop 11.* Washington, DC, USA: IEEE Computer Society, 2005, p. 233.2.

[40] S. Ghosh, M. Martonosi, and S. Malik, "Precise miss analysis for program transformations with caches ofarbitrary associativity," *SIGOPS Oper. Syst. Rev.*, vol. 32, no. 5, pp. 228–239, 1998.

[41] C. Guiang, A. Purkayastha, K. Milfeld, and J. Boisseau, "Memory performance of dual-processor nodes: comparison of IntelXeon and AMD Opteron

memory subsystem architectures," in *Proceedings for ClusterWorld Conference and Expo 2003*, San Jose, CA, June 2003.

[42] R. H. Halstead and T. Fujita, "MASA: a multithreaded processor architecture for parallel symboliccomputing," in *ISCA '88: Proceedings of the 15th Annual International Symposiumon Computer architecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 443–451.

[43] O. Hernandez, C. Liao, and B. Chapman, "A tool to display array access patterns in OpenMP programs," in *PARA'04 Workshop On State-Of-The-Art In Scientific Computing*. Springer, 2004.

[44] C. Hughes, V. Pai, P. Ranganathan, and S. Adve, "Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors," *IEEE Computer*, vol. 35, no. 2, February 2002.

[45] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: methodology andempirical data," in *36th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO-36)*, 2003.

[46] Q. Jacobson, "UltraSPARC IV processors," in *Microprocessor Forum*, 2003.

[47] S. Jarp, "A methodology for using the itanium-2 performance counters for bottleneckanalysis," HP Labs, Tech. Rep., August 2002.

[48] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and itsperformance," NASA Ames Research Center, Tech. Rep. NAS-99-011, 1999.

[49] R. Jin and G. Agrawal, "Performance prediction for random write reductions: a case studyin modeling shared memory programs," in *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS internationalconference on Measurement and modeling of computer systems*. New York, NY, USA: ACM Press, 2002, pp. 117–128.

[50] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelinedmachines," in *Proceedings of the 3rd International Conference on ArchitecturalSupport for Programming Languages and Operating System (ASPLOS)*, vol. 24. New York, NY: ACM Press, 1989, pp. 272–282. [Online]. Available: citeseer.ist.psu.edu/jouppi89available.html

[51] R. Kalla, B. Sinharoy, and J. Tendler, "IBM POWER5 chip: a dualcore multithreaded processor," *IEEE Micro*, vol. 24, no. 2, pp. 40–47, 2004.

[52] A. Kapelinkov, R. R. Muntz, and M. D. Ercegovac, "A modeling methodology for the analysis of concurrent systems andcomputations," *J. Parallel Distrib. Comput.*, vol. 6, no. 3, pp. 568–597, 1989.

[53] N. Kappiah, V. W. Freeh, and D. K. Lowenthal, "Just in time dynamic voltage scaling: Exploiting inter-node slackto save energy in mpi programs," in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005, p. 33.

[54] D. Kim, S. S. wei Liao, P. H. Wang, J. delCuvillo, X. Tian, X. Zou, H. Wang, D. Y. M. Girkar, and J. P. Shen, "Physical experimentation with prefetching helper threads on Intel'shyper-threaded processors," in *International Symposium on Code Generation and Optimization(CGO)*, San Jose, CA, March 2004.

[55] L. Kleinrock, *Queueing Systems. Volume I: Theory*. John Wiley & Sons, 1975.

[56] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded Sparc processor," *IEEE-MICRO*, vol. 25, no. 2, pp. 21–29, March/April 2005. [Online]. Available: http://csdl.computer.org/comp/mags/mi/2005/02/m2021abs.htm;http://csdl.computer.org/dl/mags/mi/2005/02/m2021.pdf

[57] D. Koufaty and D. T. Marr, "Hyperthreading technology in the netburst microarchitecture," *IEEE Micro*, vol. 23, no. 2, pp. 56–65, 2003.

[58] R. Kufrin, "PerfSuite: An accessible, open source, performance analysis environmentfor Linux," in *6th International Conference on Linux Clusters (LCI-2005)*, Chapel Hill, NC, April 2005.

[59] K. Kurihara, D. Chaiken, and A. Agarwal, "Latency tolerance through multi-threading in large-scale multiprocessors," in *Proceedings of the International Symposium on Shared Memory Multiprocessing*. Tokyo, Japan: Inf. Process. Soc. Japan, 1991, pp. 91–101. [Online]. Available: citeseer.ist.psu.edu/kurihara91latency.html

[60] M. Lee, B. Whitney, and N. Copty, "Performance and scalability of OpenMP programs on the Sun FireTME25K throughput computing server," in *WOMPAT 2004*, Houston, TX, 2004, pp. 19–28.

[61] J. Li, J. F. Martinez, and M. C. Huang, "The thrifty barrier: Energy-aware synchronization in shared-memorymultiprocessors," in *HPCA '04: Proceedings of the 10th International Symposium on HighPerformance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2004, p. 14.

[62] C. Liao, O. Hernandez, B. Chapman, WenguangChen, and W. Zheng, "OpenUH: An optimizing, portable OpenMP compiler," *Concurrency and Computation: Practice and Experience, Special Issueon CPC'2006 selected papers*, 2006(Accepted).

[63] C. Liao, Z. Liu, L. Huang, and B. Chapman, "Evaluating OpenMP on chip multithreading platforms," in *First international workshop on OpenMP*, Eugene, Oregon USA, June 2005.

[64] S.-W. Liao, A. Diwan, J. Robert P. Bosch, and A. G. M. S. Lam, "Suif explorer: an interactive and interprocedural parallelizer," in *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principlesand practice of parallel programming*. New York, NY, USA: ACM Press, 1999, pp. 37–48.

[65] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. JaneIrwin, "Exploiting barriers to optimize power consumption of cmps," in *IPDPS '05: Proceedings of the 19th IEEE International Parallel andDistributed Processing Symposium (IPDPS'05) - Papers*. Washington, DC, USA: IEEE Computer Society, 2005, p. 5.1.

[66] J. L. Lo, L. A. Barroso, S. J. Eggers, KouroshGharachorloo, H. M. Levy, and S. S. Parekh, "An analysis of database workload performance on simultaneous multithreadedprocessors," in *ISCA '98: Proceedings of the 25th annual international symposiumon Computer architecture*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 39–50.

[67] C.-K. Luk, "Tolerating memory latency through software-controlled preexecutionin simultaneous multithreading processors," in *ISCA*, 2001, pp. 40–51.

[68] B. M. Maggs, L. R. Matheson, and R. E. Tarjan, "Models of parallel computation: a survey and synthesis," in *HICSS '95: Proceedings of the 28th Hawaii International Conferenceon System Sciences (HICSS'95)*. Washington, DC, USA: IEEE Computer Society, 1995, p. 61.

[69] P. S. Magnusson, M. Christensson, J. E. andDaniel Forsgren, G. Hallberg, J. Hogberg, FredrikLarsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[70] W. Magro, P. Petersen, and S. Shah, "Hyper-threading technology: Impact on compute-intensive workloads," *Intel Technology Journal*, vol. 6, 2002.

[71] S. Manegold, "The calibrator (v0.9e), a cache-memory and TLB calibration tool," http://homepages.cwi.nl/~manegold/Calibrator/calibrator.shtml, 2004.

[72] G. Marin and J. Mellor-Crummey, "Cross-architecture performance predictions for scientific applicationsusing parameterized models," in *SIGMETRICS '04/Performance '04: Proceedings of the joint internationalconference on Measurement and modeling of computer systems*. New York, NY, USA: ACM Press, 2004, pp. 2–13.

[73] J. Markoff, "Intel's big shift after hitting technical wall," *The New York Times*, 2004.

[74] M. Martonosi, D. Brooks, and P. Bose, "Modeling and analyzing cpu power and performance: Metrics, methods,and abstractions," Tutorials Program - SIGMETRICS 2001 / Performance 2001, 2001.

[75] L. McDowell, S. J. Eggers, and S. D. Gribble, "Improving server software support for simultaneous multithreadedprocessors," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practiceof Parallel Programming(PPOPP)*, San Diego, CA, USA, June 2003, pp. 37–48.

[76] K. S. McKinley, "A compiler optimization algorithm for shared-memory multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 8, pp. 769–787, 1998.

[77] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving data locality with loop transformations," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 4, pp. 424–453, 1996.

[78] L. W. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *USENIX Annual Technical Conference*, 1996, pp. 279–294. [Online]. Available: citeseer.ist.psu.edu/mcvoy96lmbench.html

[79] S. J. Min, S. W. Kim, M. Voss, and S. I. L. R. Eigenmann, "Portable compilers for OpenMP," in *WOMPAT '01: Proceedings of the International Workshop on OpenMP Applicationsand Tools*. London, UK: Springer-Verlag, 2001, pp. 11–19.

[80] S. mirosystems, "UltraSPARC IV processor architecture overview," http://www.sun.com/processors/documentation.html, February 2004.

[81] M. S. Müller, C. Niethammer, B. C. Y. Wen, and Z. Liu, "Validating OpenMP 2.5 for Fortran and C/C++," in *Sixth European Workshop on OpenMP*, KTH Royal Institute of Technology, Stockholm, Sweden, October 2004.

[82] D. S. Nikolopoulos, "Code and data transformation for improving shared cache per-formanceon SMT processors," in *Proceedings of 5th International Symposium on High Performance Computing,ISHPC 2003*. Tokyo-Odaiba, Japan: Lecture Notes in Computer Science 2858, Springer, October 2003.

[83] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. P. J. S. Harper, and D. V. Wilcox, "Pace–a toolset for the performance prediction of parallel and distributedsystems," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, pp. 228–251, 2000.

[84] K. Olukotun, "The case for a single-chip multiprocessor," in *Intl. Conf. on Architec-tural Support for Programming Languages andOperating Systems*, 1996, pp. 2–11.

[85] "Open64 project at Rice university," http://hipersoft.cs.rice.edu/open64/, 2006.

[86] "OpenMP: Simple, portable, scalable SMP programming," http://www.openmp.org, 2007.

[87] "The OpenUH compiler project," http://www.cs.uh.edu/~openuh, 2005.

[88] "Pathscale EKOPATH compiler suite for AMD64 and EM64T," http://www.pathscale.com/ekopath.html, 2006.

[89] "Perfsuite," http://perfsuite.ncsa.uiuc.edu/, 2006.

[90] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. H. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. W. andJ. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, "The design and implementation of a first-generation CELL processor," in *the IEEE International Solid-State Circuits Conference*, 2005.

[91] A. K. Porterfield, "Software methods for improvement of cache performance on supercomputerapplications," Ph.D. dissertation, Rice University, 1989, chairman-K. W. Kennedy.

[92] V. K. Reddy, A. M. Sule, and A. V. Anantaraman, "Hyper-threading on the Pentium 4," http://www4.ncsu.edu/~vkreddy/projects/792e/792e_paper.pdf, Decemeber 2002.

[93] R. Rugina and M. C. Rinard, "Symbolic bounds analysis of pointers, array indices, and accessedmemory regions," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 2, pp. 185–235, 2005.

[94] R. H. Saavedra and A. J. Smith, "Analysis of benchmark characteristics and benchmark performance prediction," *ACM Trans. Comput. Syst.*, vol. 14, no. 4, pp. 344–384, 1996.

[95] M. Sato, S. Satoh, K. Kusano, and YoshioTanaka, "Design of OpenMP compiler for an SMP cluster," in *the 1st European Workshop on OpenMP(EWOMP'99)*, Sept. 1999, pp. 32–39.

[96] I. Sharapov, R. Kroeger, G. Delamarter, and R. C. M. Ramsay, "A case study in top-down performance estimation for a large-scaleparallel application," in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principlesand practice of parallel programming*. New York, NY, USA: ACM Press, 2006, pp. 81–89.

[97] B. Smith, "Architecture and applications of the HEP multiprocessor computersystem," in *In SPIE Real Time Signal Processing IV*, 1981, pp. 241–238.

[98] A. Snavely, N. Wolter, and L. Carrington, "Modeling application performance by convolving machine signatureswith application profiles," in *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4.2001 IEEE International Workshop on*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 149–156.

[99] D. J. Sorin, V. S. Pai, S. V. Adve, and M. K. V. D. A. Wood, "Analytic evaluation of shared-memory systems with ilp processors," in *ISCA '98: Proceedings of the 25th annual international symposiumon Computer architecture*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 380–391.

[100] SPHINX, "http://www.llnl.gov/casc/sphinx/sphinx.html."

[101] A. Spiegel, D. an Mey, and C. H. Bischof, "Hybrid parallelization of cfd applications with dynamic thread balancing." in *PARA 04 workshop on state-of-the-art in scientific computing*, 2004, pp. 433–441.

[102] L. Spracklen and S. G. Abraham, "Chip multithreading: Opportunities and challenges," in *11th International Conference on High-Performance Computer Architecture*, 2005, pp. 248–252.

[103] E. Su, X. Tian, M. Girkar, G. Haab, SanjivShah, and P. Petersen, "Compiler support of the workqueuing execution model for intel SMParchitectures," in *The Fourth European Workshop on OpenMP*, 2002.

[104] I. Sun microsystem, "Sun Studio compilers and tools," http://developers.sun.com/prodtech/cc/products/index.html. [Online]. Available: http://www.sun.com/software/products/studio/index.xml

[105] J. M. Tendler, J. S. Dodson, J. S. F. Jr., HungLe, and B. Sinharoy, "POWER4 system microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–26, 2002.

[106] X. Tian, A. Bik, M. Girkar, P. Grey, HidekiSaito, and E. Su, "Intel OpenMP C++/Fortran compiler for hyper-threading technology:implementation and performance," *Intel Technology Journal*, vol. 6, pp. 36–46, 2002.

[107] D. M. Tullsen, S. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22th Annual International Symposium on ComputerArchitecture*, 1995, pp. 392–403.

[108] M. Voss and R. Eigenmann, "Reducing parallel overheads through dynamic serialization," in *IPPS '99/SPDP '99: Proceedings of the 13th International Symposiumon Parallel Processing and the 10th Symposium on Parallel and DistributedProcessing*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 88–92.

[109] H. Wang, P. H. Wang, R. D. Weldon, S. M. E. H. Saito, M. Girkar, S. S. wei Liao, and J. Shen, "Speculative precomputation: Exploring the use of multithreading forlatency tools," *Intel Technology Journal*, vol. 6, no. 1, pp. 22–35, Feb. 2002. [Online]. Available: http://developer.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf

[110] K.-Y. Wang, "Precise compile-time performance prediction for superscalar-basedcomputers," in *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programminglanguage design and implementation*. New York, NY, USA: ACM Press, 1994, pp. 73–84.

[111] T. Wang, F. Blagojevic, and D. S. Nikolopoulos, "Runtime support for integrating precomputation and thread-level parallelismon simultaneous multithreaded processors," in *The Seventh Workshop on Languages, Compilers, and Run-time Supportfor Scalable Systems (LCR 2004)*, Houston, TX, October 2004.

[112] D. L. Whitfield and M. L. Soffa, "An approach for exploring code improving transformations," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 6, pp. 1053–1084, 1997.

[113] M. E. Wolf, D. E. Maydan, and D.-K. Chen, "Combining loop transformations considering caches and scheduling," in *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposiumon Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 274–286.

[114] S. J. V. Worley and A. J. Smith, "Microbenchmarking and performance prediction for parallel," Berkeley, CA, USA, Tech. Rep., 1995.

[115] K. Yotov, X. Li, G. Ren, M. C. andGerald DeJong, M. Garzaran, D. Padua, K. P. P. Stodghill, and P. Wu, "A comparison of empirical and model-driven optimization," in *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programminglanguage design and implementation*. New York, NY, USA: ACM Press, 2003, pp. 63–76.

[116] Y. Zhang, M. Burcea, V. Cheng, R. Ho, and M. Voss, "An adaptive OpenMP loop scheduler for hyperthreded SMPs," in *Proc. of International Conference on Parallel and Distributed Systems(PDCS-2004)*, San Francisco, CA, September 2004.