Invited Paper: A Compile-time Cost Model for OpenMP

Chunhua Liao and Barbara Chapman

University of Houston
Computer Science Department
Houston, TX, 77204 USA
{liaoch, chapman}@cs.uh.edu

Abstract

OpenMP has gained wide popularity as an API for parallel programming on shared memory and distributed shared memory platforms. It is also a promising candidate to exploit the emerging multicore, multithreaded processors. In addition, there is an increasing trend to combine OpenMP with MPI to take full advantage of mainstream supercomputers consisting of clustered SMPs. All of these require that attention be paid to the quality of the compiler's translation of OpenMP and the flexibility of runtime support. Many compilers and runtime libraries have an internal cost model that helps evaluate compiler transformations, guides adaptive runtime systems, and helps achieve load balancing. But existing models are not sufficient to support OpenMP, especially on new platforms. In this paper, we present our experience adapting the cost models in OpenUH, a branch of Open64, to estimate the execution cycles of parallel OpenMP regions using knowledge of both software and hardware. Our OpenMP cost model reuses major components from Open64, along with extensions to consider more OpenMP details. Preliminary evaluations of the model are presented using kernel benchmarks. The challenges and possible extensions for modeling OpenMP on multicore platforms are also discussed.

1 Introduction

OpenMP [21], a set of compiler directives, runtime library routines and environment variables, is the de-facto programming standard for parallel programming in C/C++ and Fortran on shared memory and distributed shared memory systems. There is an increasing trend to combine OpenMP with MPI to take full advantage of medium and large-scale computers consisting of clustered shared memory parallel processors (SMPs). It is also considered a promising candidate for exploiting emerging multicore processors.

However, getting high performance from OpenMP is a non-trivial task. On traditional SMPs, the speedup of codes using OpenMP relies on many factors, including the available parallelism in applications and the manner in which it is exploited, compiler optimizations, runtime support, data layout, operating system noise, and workload balancing and so on. The introduction of multicore processors poses considerable challenges to the OpenMP application developer, since these processors differ from the simple symmetric view of computational resources assumed in OpenMP. Moreover, they also greatly differ from each other in terms of the nature of hardware resource sharing, inter-core connections and supported logical threads per core.

Cost models[17] are a class of low level models to reflect the detailed execution characteristics of computer systems and estimate the cost, mostly the time, of executing applications using such systems. Many compilers and runtime libraries have an internal cost model that helps evaluate compiler transformations and thus guides the compiler in its optimization process, guides adaptive runtime systems, and helps achieve load balancing. However, existing cost models are too simple to be sufficient for the compilation of OpenMP, especially for new multicore platforms.

In this paper, we presents our efforts to create an efficient and accurate cost model of OpenMP based on the cost models in the OpenUH[14] compiler. With extensibility in mind, our model consists of a set of submodels accounting for different factors: processors, memory hierarchy, as well as parallel overheads. At compile time, our model uses knowledge of both software and hardware to estimate the execution cycles of parallel OpenMP regions. We evaluate our model with respect to its accuracy and efficiency using

This work was funded by NSF under contract CCF-0444468 and DOE under contract DE-FC03-01ER25502. NCSA at UIUC provided access to computing resources.

1-4244-0910-1/07/$20.00

microbenchmarks.

The reminder of this paper is organized as follows. In the next section, our motivation for building an OpenMP cost model are given. Section 3 presents an overview of OpenUH and its cost models. Our adaptation of OpenUH's cost models is described in Section 4, followed by preliminary results in Section 5. Section 6 surveys related work. Finally, Section 7 concludes by outlining future work.

2 Motivation

Cost modeling is important in many aspects of current OpenMP research, especially those related to the challenge of adapting OpenMP for emerging multicore platforms. An accurate OpenMP cost model is indispensable for performance modeling and prediction for modern parallel applications. It can help to guide compiler transformations, to improve adaptive runtime systems targeting new platforms, and to facilitate load balancing of MPI using OpenMP.

Multicore processors have entered the mainstream. A typical multicore processor has several physical processor cores integrated into one chip (Chip MultiProcessing, CMP), sometimes combined with hardware-supported simultaneous (SMT) or fine-grained/coarse-grained multithreading. There is great variety in multicore designs, especially in the amount and nature of resource sharing between threads on a given system. For example, L2 cache could be dedicated to each core or fully shared among them. Each core might support multiple threads using simultaneous/interleaved multithreading. As a result, emerging SMPs using multicore processors provide opportunities for exploitation of shared features, impose challenges arising from resource contention, and require that attention be paid to scalability.

Though a promising candidate for exploiting multicore platforms, OpenMP cannot readily meet the new challenge. As a simple fork-join parallel programming model, it was primarily designed for flat, uniform access shared-memory space (UMA) systems and for relatively modest thread counts. Initial observations [15] indicate some inefficiencies in current OpenMP on multicore platforms. For example, default strategies that make good sense on a flat SMP, such as using the maximum number of threads, an arbitrary thread-processor mapping, and static block scheduling of loops, may no longer be appropriate.

One of the major approaches to address the challenges posed by new platforms is to use empirical search. For example, Zhang et. al.[34] proposed an empirical runtime search for choosing the right number of threads and a good scheduling method for iterative applications on SMPs with hyper-threading. 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 variants such as different chunk sizes, thread placement and bindings.

On the other hand, SMP clusters dominate the high performance computing market today. OpenMP is being increasingly used with MPI on them to provide fine-grain parallelism and to improve load balancing in MPI code. A key problem here is to decide the "right" number of OpenMP threads for each MPI process. Existing simple heuristics[8, 29] for estimating execution time of OpenMP code portions are quite limited in their ability to provide accurate predictions.

Existing compile-time OpenMP cost models are simplistic in terms of functionality and applicability. For instance, to decide if parallelization is profitable, a simple heuristic based on the size of the loop body and iteration space was used in SUIF [16]. Another simple fork-join model [30] was used to derive a threshold for dynamic serialization of OpenMP. Neither consider sufficient OpenMP details for more serious usage.

An accurate and efficient cost model of OpenMP is urgently needed today for advancing research and development in OpenMP. We believe it can be widely used to complement existing solutions for the challenges imposed by new architectures and complex programming models.

3 OpenUH and its Cost Models

The OpenUH compiler[14], a branch of Open64 [1], is an optimizing and portable open-source OpenMP compiler for C/C++ and Fortran 90 programs. It is a complete optimizing 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 IR-to-source tools. As depicted in Fig. 1, OpenUH reuses major components of the Open64 infrastructure: the frontends, the interprocedural analyzer (IPA) and the middle-end/backend, which is further subdivided into the loop nest optimizer (LNO) including an auto-parallelization option (APO), global optimizer (WOPT), and code generator (CG). The IR of OpenUH (and also Open64) is named WHIRL, which has five levels to facilitate different compiler analyses and optimizations from high-level language-dependent ones to low-level machine-dependent optimizations, and anything between. Among its many enhancements, OpenUH has also incorporated features from other major branches[22, 23, 7] of Open64.

OpenUH handles OpenMP in several steps (see [14] for details). First, the source code is parsed by the appropriate extended language frontend and translated into WHIRL IR with OpenMP pragmas. Then a phase named OMP_Prelower preprocesses OpenMP pragmas for semantic checking and simplifications. Most OpenMP transformations are conducted in the LOWER_MP phase,

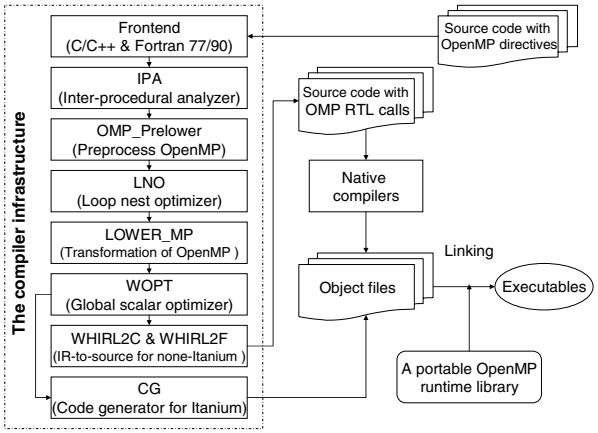

Figure 1. The OpenUH compiler

where WHIRL with OpenMP pragmas is translated into WHIRL representing multithreaded code with corresponding OpenMP runtime library calls. The remainder of the process depends on the target machine: for Itanium platforms, the code generator derived from 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 after the middle end to preserve valuable optimizations from IPA, LNO and WOPT. In this case, a native C or Fortran compiler needs to be invoked on the target platform to complete the compilation of the multithreaded C/Fortran code. In both native and source-to-source compilation models, a portable OpenMP runtime library is used to support the execution of OpenMP programs.

3.1 Cost Models

OpenUH includes a set of cost models inherited from Open64's loop nest optimizer(LNO) [32] 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 ones that are eligible to be transformed into perfect ones. The compiler uses its cost models to choose a combination of different loop level optimizations, including transformations such as arbitrary loop interchange, tiling and outer loop unrolling. The cost model may also guide automatic parallelization. There are three major models: the Processor model, Cache model and Parallel model. We briefly describe each of them below:

The processor model(see Fig.2) is mainly used to estimate the CPU cycles needed to execute one iteration of a SNL loop, without considering latencies from the memory hierarchy. The processor cycles ($Machine_c\_per\_iter$) are calculated based on cycles from FP and ALU units(OP_c), memory units(MEM_ref_c), and issue units($Issue_c$). The compiler uses a preset scheduling table to map the WHIRL IR into target machine instructions. The instructions are further associated with processor resources (FP, ALU, Branch etc.) and 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 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.

The processor model also takes into consideration the latency involved for instructions/memory operation dependencies, which are significant for processor stalls. The assumption is that the loop modeled 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, which represent the number of bytes of data references in cache by a given instruction. Instruction footprints are further accumulated to loop level footprints. The unused data residing in the same cache lines as the referenced data is also considered as part of footprints as 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 Fig.3) is accumulated from all levels of cache hierarchy using $miss * penalty$ for simplicity.

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 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. Fig.3 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.

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

Figure 2. Equations of Processor Model

$$
\begin{aligned}
Total_c &= Machine_c + TLB_c + Cache_c + Loop\_overhead_c + Parallel\_overhead_c \\
Machine_c &= Machine_c\_per\_iter * Num\_loop\_iter/Num\_threads \\
TLB\_miss &= Num\_array\_ref - TLB\_entries, if(Num\_array\_ref - TLB\_entries > 0) \\
TLB_c &= TLB\_miss\_penalty * TLB\_miss \\
Cache_c &= \sum_{i=1}^{Levels} (Clean\_footprint_i * Clean\_penalty_i + Dirty\_footprint_i * Dirty\_penalty_i) \\
Loop\_overhead_c &= Loop\_overhead\_per\_iter_c * Num\_loop\_iter/Num\_threads \\
Parallel\_overhead_c &= Parallel\_startup_c + Parallel\_const\_factor_c * Num\_threads
\end{aligned}
$$

Figure 3. Equations of Parallel Model and Cache Model

4 Modeling OpenMP

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

Extending the parallel model to model OpenMP has been proved to be mostly straightforward, since 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. The extensions cover more details of the OpenMP programming API and are depicted in Fig.4. We show only the C/C++ aspects here; our model for OpenMP in Fortran programs is similar.

Our OpenMP cost model focuses on a parallel region at a time, which in turn may contain multiple worksharing regions (`omp for`, `omp section` or `omp single`), and multiple synchronization constructs (`omp master`, `omp critical`, etc.). 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 fork-join overheads. $Thread_i\_exe\_j_c$ is in turn the sum of enclosed worksharing and synchronization costs. Taking the maximum execution time between synchronization points of all participating threads into account ensures the overall accuracy and exposes possible load imbalance. Also, our model considers scheduling overhead cycles ($Schedule_c$) and chunk size, aiming to model different scheduling policies. The rest of the terms in the model's equations directly derive from the original equations of the parallel model in OpenUH, including $Machine_c\_per\_iter$, $Cache_c$, and $Loop\_overhead_c$, etc.

There are several implementation details worth mentioning. First of all, the original parallel model only worked for sequential SNL loops. Thus a new IR traversal phase was added to locate all OpenMP parallel regions and to apply our OpenMP cost model to each of them subsequently. In addition, 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-join overhead, scheduling overheads,etc.) used in the OpenMP model can be obtained or derived from microbenchmarks[5, 28]. 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.

5 Preliminary Results

We provide some early results of evaluating our cost model for OpenMP in this section. The benchmark used was a classic matrix-matrix multiplication(MMM) kernel (Listing 1), which has also been widely used in previous research[32, 33] due to its importance in scientific computation. We selected three square array sizes (500×500,

$$
\begin{aligned}
Parallel\_region_c &= Fork_c + \sum_{j=1}^{m}[maximum(Thread_0\_exe\_j_c, ..., Thread_{n-1}\_exe\_j_c)] + Join_c \\
Thread_i\_exe\_j_c &= Worksharing\_c + Synchronization\_c \\
Worksharing\_c &= Parallel\_for_c / Parallel\_section_c / Single_c \\
Synchronization\_c &= Master_c / Critical_c / Barrier_c / Atomic_c / Flush_c / Lock_c \\
Parallel\_for_c &= Schedule\_times * (Schedule_c + Loop\_chunk_c + Ordered_c + Reduction_c) \\
Loop\_chunk_c &= Machine_c\_per\_iter * Chunk\_size + Cache_c + Loop\_overhead_c
\end{aligned}
$$

Figure 4. Equations of Cost Model for OpenMP

1000×1000, and 1500×1500) to study the impact of different input data sets on our model. Our test platform was COBALT, an SGI Altix system at NCSA. COBALT is a cc-NUMA platform with a total of 32 1.5 GHz Itanium 2 processors and 256 GB memory.

Listing 1. Matrix multiplication in OpenMP

```
#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];
```

Major machine and other modeling parameters are shown in Table 1 and 2. They were obtained in several ways: vendor manuals, system information from an open-source performance analysis toolset named PerfSuite[24], and microbenchmarks such as LMbench[20] and EPCC[5]. We used OpenUH to compile all versions of the MMM kernel with compilation option `-mp -O3` because our OpenMP cost model assumes that optimizations occur. The executables were run from 1 to 8 threads and PerfSuite was used to collect performance metrics.

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 1. Major processor/parallel parameters

Fig.5 shows both modeled and measured CPU cycles of all versions of the MMM kernel. The accuracy of the OpenMP cost model was depicted in Fig. 6 using the relative difference ratio $((Modeled_c - Measured_c)/Measured_c)$ between modeled cycles and measured cycles. The results of modeling OpenMP with `schedule` clause is given in Fig.7 using array size 1000×1000 with 4-thread execution. Only `static` scheduling results are shown because `dynamic` and `guided` scheduling have very similar results. It is interesting that the compiler optimized the innermost two level loops of the MMM kernel extensively: loop interchanging, tiling at k loop, as well as unrolling at j loop.

	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 2. Major memory hierarchy parameters

It is obvious that our current model leaves plenty of room for tuning and improvements. Many factors contribute to the large difference ratio between modeling and measuring results: compile-time cost modeling solely relying on static analysis, an input high-level IR which is quite different from the final assembly code, system noise when measuring multithreaded execution, as well as ignored sources of execution cost such as instruction cache misses, coherence cache misses, and bus contentions. However, our simple model is still able to capture the relative performance of different chunk sizes of static scheduling, which would often give enough hints for guiding OpenMP compilation and runtime tuning.

Finally, we measured the amount of time spent on OpenMP cost modeling of the MMM kernel. On average, only 0.079 second was needed when invoking the OpenMP cost model for the first time, which accounted for 1.25% $(0.079s/6.33s)$ of the total compilation time of the kernel. And the time spent on additional evaluations using different threads was negligible since they could reuse many results of the initial modeling. Thus our model is efficient enough for frequent usage within the compiler.

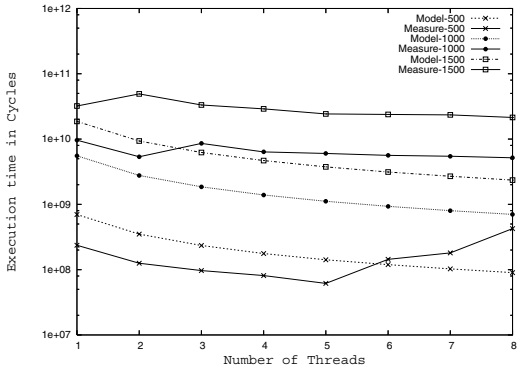

Figure 5. Modeling vs. Measuring

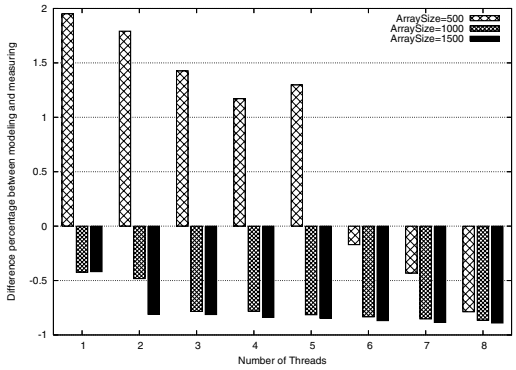

Figure 6. Accuracy of Modeling

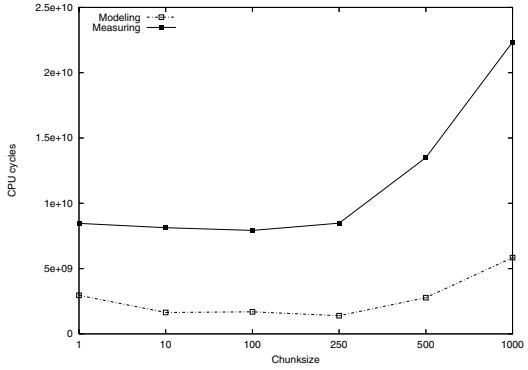

Figure 7. Modeling schedule(static,n)

6 Related Work

Building analytical models that incorporate knowledge of both hardware and software has been a research topic since the early days of computing, because they can be widely used to evaluate different computer system designs, to select compiler transformations, and also to tune application performance. We only mention a few of them below since an exhaustive list is beyond of the scope of this paper.

Numerous compiler cost models for computation and memory access have been proposed, especially for guiding loop transformations [32]. Wang [31] presents a cost model for superscalar processors considering multiple function units and instruction dependency. His work also maps high level language constructs to low level machine operations using cost tables. Ferrante et al. [10] 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 [19] is based on equivalence classes of array references showing temporal and spatial locality. Ghosh et al. [11] introduces cache miss equations based on a system of linear Diophantine equations from a reuse vector. A more recent study by Yotovo et. al.[33] 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.[25] 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 control flow graph edge frequency and memory reuse distance obtained from binary profiling of benchmarks, Marin et. al.[18] apply approximation functions to build parameterized performance models. Both of their models are limited to sequential programs only. Snavely et. al.[26] 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. Their model only targets memory-intensive applications.

Models [4, 27] are also used to estimate the performance of parallel programs. Some of them [9] are at a high level, proving qualitative insights. Some are based on task graphs [3]. Several integrated performance frameworks [2] also exist. However, they are often not lightweight and versatile enough for compile-time usage. There are several compile-time parallel models: a simple heuristic based on the size of the loop body and iteration space is used in SUIF [16] to decide if parallelization is profitable. A simple fork-join model [30] is used to derive a threshold for dynamic serialization of OpenMP. Considering coherence misses, lock time and memory contention, Jin et al. [13] presents an analytical model for SMPs to select optimizations for reductions, but not for OpenMP applications.

To the best of our knowledge, there are very few models for multicore/multithreaded platforms. Chandra et. al. [6] model the extra L2 cache misses due to inter-thread contention on a chip multi-processor architecture using stack distance or circular sequence profile. Their mod-

els are limited to co-scheduled threads from different sequential benchmarks, and are thus not directly applicable to OpenMP threads.

As to the accuracy, profiling or microbenchmarking-based models[25, 26] usually have good accuracy: the average difference ratio between predicted and measured performance ranges from $\pm 10\%$ to $\pm 30\%$. Models[12, 3, 6] using mathematical equations extensively have much wider difference ranges (from a few percentage to even hundreds percentage) or no accuracy information[31] was provided at all. Most compile-time cost models[32, 19] have not been directly validated for accuracy, only showing performance improvement using model-driven optimizations.

7 Discussion and Future Work

Building cost models is challenging, but valuable. The major difficulty is the deep knowledge needed about hardware, software and their internal and external interactions, and the trade-off between accuracy and efficiency.

Getting machine profiles is not easy. The vendor manuals cannot cover each variant of their hardware and results of microbenchmarks often do not agree with each other. The popularity of multicore platforms demands 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 resource-sharing features in multicore processors. As for user applications, compile-time models usually have limited ability to foresee the impact of different input data sets. The complex, implementation-dependent optimizations conducted on the applications make it difficult for models to predict the final form of the code. Modeling parallel programs on multicore platforms is much harder, considering the exponentially increased execution possibilities and non-deterministic behavior of concurrent executing threads. Taking OpenMP as an example, an accurate cost model has to consider additional factors like the activated thread context and thread-processor mapping and bindings.

The modeling of interactions among hardware/software components is a another tough call. The complexity of modern out-of-order, superscalar processors and multiple levels of non-blocking memory hierarchy already impose numerous challenges. 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. The mainstream modeling approach today is still to consider one component at one time and combine their results in simple ways, which could lead to significant inaccuracy. Again, multicore platforms introduce more variants in studying hardware/software interactions.

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

On the other hand, 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.

Our future work can follow several directions. One is to extend our cost model for multicore platforms as well as improving its accuracy on traditional SMPs. Exploring portable and reusable ways to build cost models is another major task. Using simulators in addition to real platforms could enable the exploration of more machine configurations and thus help revise and validate our model. Finally, we are considering parameterizing all the models, in order to widen its applicability for both static compilers and dynamic runtime systems. This may enable it to support future combinations of simulation, profiling, measurement, runtime monitoring and analytical modeling.

References

[1] The Open64 compiler. `http://www.open64.net`, 2006.

[2] V. S. Adve, R. Bagrodia, J. C. Browne, E. Deelman, A. Dube, E. N. Houstis, J. R. Rice, R. Sakellariou, D. J. Sundaram-Stukel, P. J. Teller, and M. K. Vernon. Poems: End-to-end performance design of large parallel adaptive computational systems. *IEEE Trans. Softw. Eng.*, 26(11):1027–1048, 2000.

[3] V. S. Adve and M. K. Vernon. Parallel program performance prediction using deterministic task graph analysis. *ACM Trans. Comput. Syst.*, 22(1):94–136, 2004.

[4] 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 Parallel architectures and compilation techniques*, pages 194–203, Manchester, UK, UK, 1995. IFIP Working Group on Algol.

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

[6] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA '05: Proceedings of the 11th International*

Symposium on High-Performance Computer Architecture, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

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

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

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

[10] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 328–343, London, UK, 1991. Springer-Verlag.

[11] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. *SIGOPS Oper. Syst. Rev.*, 32(5):228–239, 1998.

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

[13] R. Jin and G. Agrawal. A methodology for detailed performance modeling of reduction computations on smp machines. *Perform. Eval.*, 60(1-4):73–105, 2005.

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

[15] 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.

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

[17] 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 Conference on System Sciences (HICSS'95)*, page 61, Washington, DC, USA, 1995. IEEE Computer Society.

[18] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 2–13, New York, NY, USA, 2004. ACM Press.

[19] K. S. McKinley. A compiler optimization algorithm for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 9(8):769–787, 1998.

[20] L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.

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

[22] Open research compiler for Itanium processor family. `http://ipf-orc.sourceforge.net`, 2005.

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

[24] Perfsuite. `http://perfsuite.ncsa.uiuc.edu/`, 2006.

[25] R. H. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Trans. Comput. Syst.*, 14(4):344–384, 1996.

[26] A. Snavely, N. Wolter, and L. Carrington. Modeling application performance by convolving machine signatures with application profiles. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 149–156, Washington, DC, USA, 2001. IEEE Computer Society.

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

[28] SPHINX. http://www.llnl.gov/casc/sphinx/sphinx.html.

[29] 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*, pages 433–441, 2004.

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

[31] K.-Y. Wang. Precise compile-time performance prediction for superscalar-based computers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 73–84, New York, NY, USA, 1994. ACM Press.

[32] 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 symposium on Microarchitecture*, pages 274–286, Washington, DC, USA, 1996. IEEE Computer Society.

[33] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 63–76, New York, NY, USA, 2003. ACM Press.

[34] 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.