A Compiler Framework for Feedback-Directed Parallelizing Programs on CMP

Yuanming Zhang, Kanemitsu Ootsu, Takashi Yokota, Takanobu Baba

Department of Information Science, Faculty of Engineering, Utsunomiya University

1 INTRODUCTION

Recently, with the increasing difficulties in achieving higher performance by growing clock frequencies on uniprocessor, the microprocessor industry has paid much attention to Chip Multiprocessor (CMP), which represents an evolutionary change in high performance computing [1]. CMP can boost the performance of multi-programmed or multi-threaded programs, while it cannot directly boost the performance of a large number of existing sequential programs.

To improve the performance of sequential programs on CMP, some techniques have been put forward. Prefetching is one of the techniques, which uses the user-level thread (helper thread) running on one idle core to prefetch the data before it is needed by the main thread [2]. This technique can improve the performance of sequential programs by reducing the large cache latency. However, this technique only focuses on cache miss and has limited increment. Decouple software pipelining (DSWP) is another technique, which extracts non-speculative threads from sequential programs [3]. This technique is valid for sequential programs, while it needs an additional hardware, a synchronization array, to support the inter-thread communication. In commercial CMP, this synchronization array is unavailable.

The objective of our research is to parallelize sequential programs automatically on commercial CMP architecture without additional hardware support. By this method, the performance of most sequential programs can be improved. The remainder of this paper is organized as follows: In next section, we will analyze the challenges of parallelizing sequential programs on commercial CMP and our approaches to eliminate them. Section 3 describes our feedback-directed compiler framework. Section 4 provides our conclusion.

2 CHALLENGES OF PARALLELIZING ON CMP

2.1 Communication Latency

Almost all general purpose processor chips are moving to Chip multiprocessor, including the Panther chip from Sun, Power 4/5 chips from IBM, Opteron chip from AMD, Woodcrest and Montecito chips from Intel. One important character of these CMPs is whether they have shared cache.

For the CMP that has no shared cache, although the separation of the cache make it possible to have dedicated access paths to the caches and eliminates contention and capacity pressure at the cache and this can offer up performance increase for multiple applications, it has no benefit to multi-threaded programs. The inter-thread communication latency will be much long on this kind of CMP, as the communication takes place in the main memory. For the CMP that has shared cache, the shared cache can increase the efficiency of cache to processor data transfer and core to core communication. With the shared cache, the data can be stored in one place that each core can access [1].

For above reasons, to reduce the inter-thread communication latency and boost the performance of multi-threaded programs, we assume the CMP has a shared cache. Our research is based on this kind of CMP.

2.2 Non-Speculative Multithreading

Speculative multithreading has been proved to be a promising method to extract parallelism from sequential programs. The efficiency of this execution model strongly depends on the performance of the control and data speculation techniques. Several multithreaded architectures, which provide support for thread-level speculation, have been proposed, such as the Multiscalar architecture, the SPSM architecture, and the Superthreaded architecture. These microarchitectures provide multiple contexts and appropriate mechanisms to forward values produced by on thread and consumed by another, and also provide hardware support to handle recovery in the case of misspeculation.

On commercial CMP, there is no special hardware to provide support for speculation. All the thread states and forwarded data have to be stored in the shared cache. For this reason, we consider to use the non-speculative multithreading parallelism. This parallelism mechanism does not need additional hardware support, and also provide an orthogonal method to parallelize sequential programs.

These non-speculative threads extracted from sequential programs run concurrently and do different parts of computation. If necessary, they communicate on some points. The communication points are the inter-thread data dependent points. In essence, these threads are cooperated threads.

The count of communication points is the count of data dependences between threads. A large count of communication points will add the communication cost, and is no benefit to the performance. To reduce the communication cost, the extracted threads are composed of strongly connected basic blocks (SCCs) [3]. The SCCs have the smaller dependences with other basic blocks, and this can lower the communication cost.

2.3 Thread Region Selection

Now, our research mainly focuses on loops, because 1). Loops consume most of programs execution time, and 2). Loops are the potential area for multithreading. In general, any loop can be parallelized. However, when sequential programs are partitioned into multithreaded programs, some cost is necessary, such as cost of creating threads, managing threads and cost of inter-thread communication and synchronization. To amortize the multithreading cost, the loops partitioned must be long running time.

Our aim is to extract non-speculative threads from loops. Moreover, these threads have fewer data dependences between each other. To achieve this aim, we think two kinds of loops are suitable for candidates. One is the nesting loop. In nesting loop, the boundary of the inner loop is clear to the boundary of the outer loop, and they maybe have fewer communication points. Another is the loop that contains procedure calls. The procedure is well strongly connected and is also long running time. On the other hand, if the procedure is too big, it can be partitioned into more threads further.

3 COMPILER FRAMEWORK

3.1 Overview

The layout of the compiler framework is shown in Fig.1. Our research is based on GCC open source compiler. The sequential program is first converted into static single assignment (SSA) form by GCC front-end and Gimpiler. The SSA form is a new intermediate representation (IR) in GCC, which is both language and target independent and allows high level analyses and transformations. The GCC analyzer is the available functions in GCC, such as the alias analysis.

In this framework, there are two important components: profiler and thread generator. The profiler collects profiling data, such as size of the loop, data dependence information and control dependence information. These profiling data, as feedback data, will be used by thread generator to partition sequential program.

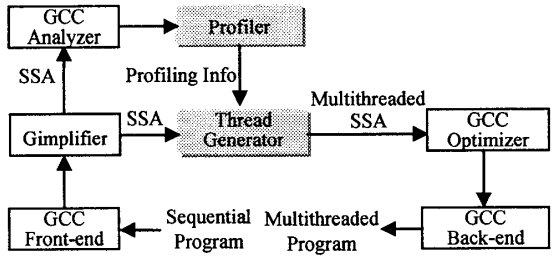

Fig.1 Compiler Framework for Feedback-Directed Parallelizing

The thread generator partitions the sequential program one procedure at a time. Firstly, the procedure is analyzed and all loops within the procedure are built into an internal form, which holds the CFG information. The CFG information includes the statements, basic blocks and edges of the loop. Secondly, the data dependences and control dependences between basic blocks are analyzed, and a dependence graph is built, which is the foundation for extracting threads.

Thirdly, by analyzing the dependence graph, the thread generator heuristically searches and selects SCCs from the graph. Once the SCCs are selected, they will be moved outside of the loop as the computation contents of a new thread. If possible, other more SCCS can be selected and moved outside of the loop as the computation contents of other new threads. The remainder of basic blocks forms the computation contents of the main thread. In this process, some instructions are inserted for data communication between threads.

The output of the thread generator is multithreaded SSA. The multithreaded SSA can be optimized by various optimization passes available in GCC, such as dead code elimination, and loop invariant motion. At last, the multithreaded SSA is compiled into multithreaded executable program by the back-end of GCC.

3.2 Thread Spawning Mechanism

In our system we use the POSIX thread as the new created thread. At the beginning of the program, we may create some POSIX threads in advance for later use. When the threads are created, they sleep before assigned task. If one thread is needed to do computation, the compiler forwards the starting address and necessary dependent data to it, and then activates it. These threads run in parallel and communicate on one data buffer. The data buffer will be introduced in detail later.

If there is no available thread, the compiler can create new threads. This depends on how many parts the loop are partitioned. These threads compute different parts of the loop, and cooperate with each other.

3.3 Thread Communication Model

In our compiler framework, a special communication cache, queued data buffer, is designed. It implements queued semantics, and provides uniformed operation semantics. All the dependent data between threads are written and read from the data buffer.

The queued data buffer structure is shown in Fig.2. The row indicates the available buffers. The dependent data is written and read from these buffers. The column indicates the available elements for each buffer. For each buffer, there are two pointers. One pointer points to next empty elements, called writing pointer (WP) which is used by producer thread. Another pointer points to current unused data, called reading pointer (RP) which is used by consumer thread. For example, in the first buffer, the WP points to the last empty element, and the RP points to the second element that contains the current unused data.

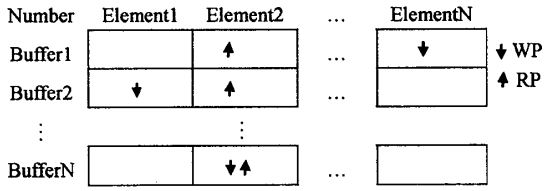

Fig.2 Queued Data Buffer Structure

With this queued data buffer, the threads can implement pipelining execution model by writing (reading) dependent data into (from) the buffer. The producer thread writes data into the buffer and the consumer thread reads data from the buffer. A thread only stalls when the queued data buffer is either full or empty. In the last buffer, as there is no empty element, the producer thread will stall for empty elements.

4 CONCLUSION

This paper presents a compiler framework for parallelizing sequential programs on commercial CMP. We analyzed the main challenges of doing that and gave our approaches to address them. Based on the feedback profiling data, the compiler extracts non-speculative threads from loop. These extracted threads cooperate together, run concurrently and communicate on a queued data buffer. Based on the buffer, the threads can execute in pipelining execution model.

ACKNOWLEDGMENTS

This research was supported in part by Grant-in-Aid for Scientific Research ((B)18300014, (C)19500037) and Young Scientists ((B)17700047) of Japan Society for the Promotion of Science (JSPS), and by Eminent Research Selected at Utsunomiya University.

REFERENCES

[1] Pawe Gepner, Micha F. Kowalik. Multi-Core Processors: New Way to Achieve High System Performance. Proceedings of the International Symposium on Parallel Computing in Electrical Engineering, 2006.

[2] Jiwei Lu, Abhinav Das, Wei-Chung Hsu, Khoa Nguyen. Dynamic Helper Threaded Prefetching on the Sun UltraSPARCR CMP Processor. Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, 2005.

[3] Guilherme Ottoni, Ram Rangan. Automatic Thread Extraction with Decoupled Software Pipelining. Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, 2005.