Compiling for Speculative Architectures[⋆]

Seon Wook Kim and Rudolf Eigenmann

School of Electrical and Computer Engineering
Purdue University, West Lafayette, Indiana

Abstract. The traditional target machine of a parallelizing compiler can execute code sections either serially or in parallel. In contrast, targeting the generated code to a speculative parallel processor allows the compiler to recognize parallelism to the best of its abilities and leave other optimization decisions up to the processor's runtime detection mechanisms. In this paper we show that simple improvements of the compiler's speculative task selection method can already lead to significant (up to 55%) improvement in speedup over that of a simple code generator for a Multiscalar architecture. For an even more improved software / hardware cooperation we propose an interface that allows the compiler to inform the processor about fully parallel, serial, and speculative code sections as well as attributes of program variables. We have evaluated the degrees of parallelism that such a co-design can realistically exploit.

1 Introduction

Parallelizing compilers have made significant progress over the past ten years. While they could exploit parallel machines effectively in only two out of twelve scientific/engineering programs in 1990 [1], today they are successful in half of these programs [2, 3]. Still, there remains a tremendous potential to be exploited. For example, today's compilers are still limited in exploiting parallelism in most C programs. Speculative architectures [4–8] can potentially overcome these limitations.

This paper makes two contributions towards our goal of developing advanced compiler technology that exploits speculative processors. First, starting from a simple code generator for Multiscalar architectures [4], we have improved the methods for selecting speculative tasks (i.e., code sections that will be executed speculatively in parallel) [?]. Second, we propose an improved software/hardware interface that allows the compiler to inform the processor about code sections that are provably parallel or serial and those that are worth parallelizing speculatively at runtime. We have identified these code sections in several programs and have evaluated the degree of parallelism that can be exploited.

[⋆] This work was supported in part by NSF grants #9703180-CCR and #9872516-EIA. This work is not necessarily representative of the positions or policies of the U. S. Government.

Table 1. Simulation results on the Multiscalar architecture. The table shows the improvements of three Perfect Benchmarks due to our enhanced task selection techniques. We measured IPC (instructions per cycle), speedup (the ratio of execution time on one processor to that on four processors), and load imbalance. Each processor has a two-way superscalar architecture. For comparison, the table shows the real speedups obtained by the automatically parallelized programs on a 4-processor Sun UltraSPARC server.

		Multiscalar					
		Original			Improved		
Benchmarks	Sun	IPC	Speedup	Load imbalance	IPC	Speedup	Load imbalance
`TRFD`	2.42	2.28	1.42	38.4%	3.27	1.97	9.5%
`BDNA`	2.22	1.55	1.48	49.0%	1.62	1.55	5.7%
`ARC2D`	4.07	1.69	1.49	38.7%	2.42	2.13	6.6%

2 Informed Task Selection

As a starting point of our project we have measured the performance of a number of programs on a Multiscalar architecture simulator. We expected to see good parallel efficiency on numerical programs, such as the Perfect Benchmarks. However we have found that, on a machine with 4 processing units (PU) both the speedup relative to a 1-PU architecture and the number of instructions issued per machine cycle (IPC) are limited. In fact, the speedup is much less that the one measured for the compiler-parallelized programs executed on a 4-processor Sun UltraSPARC server. Table 1 shows these measurements.

We have found that one reason for the limited performance is the simple task selection scheme applied in the compiler: Each basic block is dispatched to the next PU for parallel execution. However, each loop iteration consists of at least two basic blocks - the actual loop body and a small block containing the branch code to the next iteration and loop end. It is obvious that these two basic blocks are not only intrinsically sequential but also very different in length, leading to load imbalance. Combining the two basic blocks requires knowledge of the program's loop structure. We have modified the compiler (a modified version of GCC) so that it performs this transformation. It results in improvements of up to 55% in speedup, as shown in Table 1.

Additional improvements can be made if the compiler considers data dependence information. For example, loop iterations which the compiler can prove as dependent can be combined into a single task. The compiler/architecture interface, described in the next section, will enable this and other optimizations.

3 Compiler / Hardware Co-design

Our goal is a scenario where the compiler detects parallelism to the maximum of its abilities and leaves the rest up to the speculative hardware. More specifically, the parallelizing compiler will detect and mark parallel loops, which will then be executed by the processor in fully-parallel mode. Fully parallel execution will

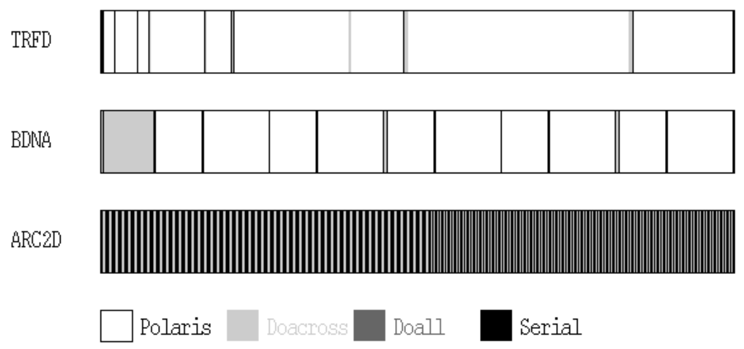

Fig. 1. Maximum parallelism that can be exploited in the programs `TRFD`, `BDNA` and `ARC2D`. Compile-time parallel sections can be exploited without speculative hardware support. Runtime Doall regions can be detected as fully parallel by the hardware. Doacross sections are partially parallel regions, also detectable at runtime.

not involve any speculative data tracking mechanism, saving overhead and obviating the need to reserve speculative data space. Data space is saved because, in speculative operation, all written data would have to be buffered until the task is no longer speculative. This can severely restrict the size of allowable speculative tasks. Further overhead can be saved by marking serial (i.e, dependent) code sections. The code generator will create a single task from such code sections, avoiding task squashes as a result of dependence violations at runtime. Only code sections that are neither parallel nor serial will become speculative tasks. To enable this operation our basic compiler/architecture interface will

- define (the boundaries of) serial, parallel and speculative tasks,
- define variable attributes as shared dependent, shared independent, speculative, private, or reduction operation. Speculative variables are those whose accesses cannot be disambiguated at compile time. Private variables are those that the compiler can prove to be written before read within a task. Reduction variables are only used as results of reduction operations.

Note that this interface requires a minimal extension of the Multiscalar architecture: speculation can be suspended, such that the machine behaves like a multiprocessor. This mode will be employed during the execution of fully parallel tasks.

We have evaluated performance opportunities of this scenario. The bars in Figure 1 show sections in the execution of `TRFD`, `BDNA` and `ARC2D` that can be detected as parallel at compile time (using the Polaris parallelizer) and at runtime, respectively. Runtime parallelism is split into sections that are fully parallel (doall) and sections that need synchronization (doacross). We have performed this analysis using the Max/P tool [?]. The figure shows that there are both significant regions of compile-time and runtime parallelism. Developing efficient mechanisms to exploit this potential is the object of our ongoing work.

4 Conclusion

We have presented a new compiler/architecture model for speculative processors. Parallel tasks that can be detected at compile time are run in non-speculative, multiprocessor mode. Program sections that may have dependences are parallelized speculatively by the architecture. From the architecture's point of view, non-speculative parallel execution can save significant overhead and can increase the parallel task size. We are currently completing the integration of the Polaris parallelizing compiler with a GCC-based code generator, which will provide the proposed architecture interface. The code generator will use high-level information from the optimizing preprocessor to generate the information required in the interface and to improve the quality of the generated code. In an initial study to improve the selection of speculative parallel tasks we have already achieved performance gains of up to 55% in speedup.

References

1. William Blume and Rudolf Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.
2. M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, pages 84–89, December 1996.
3. W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, pages 78–82, December 1996.
4. Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. *The 22th International Symposium on Computer Architecture (ISCA-22)*, pages 414–425, June 1995.
5. Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessors. *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.
6. J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. *Proceedings of the Fourth International Symposium on High-Performance Computer Architectures*, February 1998.
7. J.-Y. Tsai, Z. Jiang, Z. Li, D.J. Lilja, X. Wang, P.-C. Yew, B. Zheng, and S. Schwinn. Superthreading: Integrating compilation technology and processor architecture for cost-effective concurrent multithreading. *Journal of Information Science and Engineering*, March 1998.
8. Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. Hardware for speculative parallelization in high-end multiprocessors. *The Third PetaFlop Workshop (TPF-3)*, February 1999.