The Structure of a Compiler for Explicit *and* Implicit Parallelism*

Seon Wook Kim and Rudolf Eigenmann

School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN
eigenman@purdue.edu

Abstract. We describe the structure of a compilation system that generates code for processor architectures supporting both explicit and implicit parallel threads. Such architectures are small extensions of recently proposed speculative processors. They can extract parallelism speculatively from a sequential instruction stream (implicit threading) *and* they can execute explicit parallel code sections as a multiprocessor (explicit threading). Although the feasibility of such mixed execution modes is often tacitly assumed in the discussion of speculative execution schemes, little experience exists about their performance and compilation issues. In prior work we have proposed the Multiplex architecture [1], supporting such a scheme. The present paper describes the compilation system of Multiplex.

Our compilation system integrates the Polaris preprocessor with the Gnu C code generating compiler. We describe the major components that are involved in generating explicit and implicit threads. We describe in more detail two components that represent significant open issues. The first issue is the integration of the parallelizing preprocessor with the code generator. The second issue is the decision when to generate explicit and when to generate implicit threads. Our compilation process is fully automated.

1 Introduction

Automatic program parallelization for shared-memory multiprocessor systems has made significant progress over the past ten years, and it has been most successful in Fortran code, as demonstrated by the Polaris and the SUIF compilers [2, 3]. However, substantial challenges still exist when detecting parallelism in irregular and non-numeric applications. One serious limitation of the current generation of compilers is that they have to provide absolute guarantees about program data dependences between sections of code believed to be parallel. Both software and hardware techniques have been proposed to overcome

* This work was supported in part by NSF grants #9703180-CCR and #9974976-EIA. Seon Wook Kim is now with KAI Software Lab, A Division of Intel Americas, Inc., Champaign, IL, seon.w.kim@intel.com.

this limitation. Multi-version codes and run-time dependence test [4, 5] are examples of software techniques, while speculative multiprocessors are proposals for hardware solutions [6–10]. The software's run-time dependence test instruments codes to track data dependences during speculative parallel execution. Speculative multiprocessors provide such speculative support in hardware.

Combining explicit parallelization and speculative execution may appear straightforward. In fact, some recent proposals for speculative architectures make this tacit assumption. However, little experience exists with combined schemes. In [1] we have presented an architecture that supports such an execution mechanism. The present paper describes the issues and the structure of a compiler generating code for such an architecture.

A compiler that supports both explicit and implicit parallelism must perform the following tasks.

Recognition of parallelism: The basis for explicit threading is the automatic or manual recognition of code sections that can execute in parallel. In this paper we consider automatic parallelization.

Selecting explicit and implicit threads: Explicit threads can be selected from the code sections identified as parallel. The remaining code may be split into implicit threads, that is, into code sections that the hardware can then execute speculatively in parallel.

Thread preprocessing: High-level transformations of the selected threads may be necessary. For example, our compiler transforms all explicitly parallel code sections such that the absence of cross-thread dependences is guaranteed.

Thread code generation: The actual code generation may include steps such as inserting calls to thread management libraries, identifying threads (e.g., adding thread header code), and setting up thread environments (e.g., defining thread-private stacks).

A number of issues arise when realizing such a compilation system. For this paper we rely on state-of-the-art technology for recognizing parallelism. Although there are many open issues in the detection of parallel code, they are the same as for automatic parallelization on today's multiprocessor systems. In our work we make use of the capabilities of the Polaris parallelizer [2].

Selecting explicit and implicit threads is an important new issue. A straightforward scheme would select all recognized outermost parallel loop iterations as explicit threads, which is the scheme chosen in today's parallelizing compilers. For the rest of the program it would select one or a group of basic blocks as implicit threads, which is the scheme used in the Wisconsin Multiscalar compiler [11]. In [12], preliminary studies showed that such a simplistic scheme would lead to suboptimal performance. Both execution schemes incur intrinsic overheads. For fully independent threads, the choice between implicit and explicit execution is important for best performance. An improved scheme must estimate overheads incurred by the two threading schemes and find the best tradeoff. We will discuss these issues and present a thread selection algorithm in Section 4.

Thread preprocessing is important in our compiler to insert code that makes runtime decisions if static thread selection was judged insufficient. Preprocessing is also necessary to guarantee the absence of data dependences among explicit threads. In Section 2.3 will will describe why code sections that are recognized as fully parallel may still incur dependences in the generated code. We use state-of-the-art compiler techniques to resolve this problem.

The actual code generation must properly identify and coordinate code sections that are to be executed as explicit or implicit threads. The start of each thread must be marked in the binary code, the thread mode (explicit or implicit) must be specified, and thread management code must be inserted.

One of the biggest implementation issues arises because the first three steps above are typically done in the preprocessor, whereas the last step is performed by the code generating compiler. Integrating the two different compilers involves expressing the results of the preprocessor's analysis in a way that can be communicated to the code generator. One also must extend the code generator to understand and use this information. We describe the interface between preprocessor and code generator in our compilation system in Section 3.

The specific contributions of this paper are as follows:

- We describe the structure and the tasks of a compiler for an architecture that supports both explicit and implicit parallelism.
- We describe techniques for integrating a parallelizing preprocessor and a code generating compiler to accomplish these tasks.
- We describe an algorithm for selecting explicit and implicit threads.

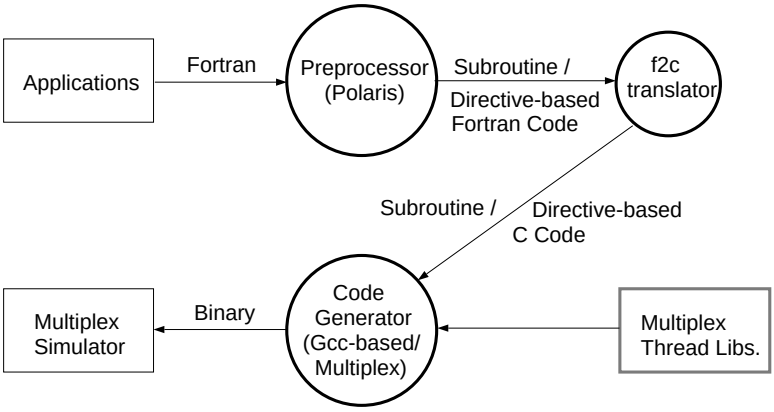
The remainder of the paper is organized as follows. Section 2 gives an overview of our compilation system in terms of the tasks identified above. The modules for which we used state-of-the-art compiler components will be briefly explained. The next two sections describe in more details the two most difficult issues we encountered. These are (1) the integration of the preprocessor and the code generating compiler (Section 3) and (2) the implicit/explicit thread selection algorithm (Section 4). We present performance results in Section 5, followed by conclusions in Section 6.

2 Overview of the Compilation System

Figure 1 gives an overview of the compilation system for our experimental architecture, Multiplex [1]. It consists of an extended version of the Polaris preprocessor [2], a modified f2c translator [13], an extended version of the Gnu C compiler [14], and Multiplex-specific thread libraries.

2.1 Recognition of Parallelism

Fortran applications are analyzed by the preprocessor to identify parallel loops. We use the existing Polaris infrastructure for this purpose. Polaris is a parallelizing preprocessor that contains advanced techniques, such for data-dependence



Compiler / Tool	Multiplex Extension	Purpose
Polaris	Thread selector pass	Select either explicit or implicit threading at compile time and runtime (multi-versioning) to minimize explicit threading overhead.
	Subroutine-based postpass	Guarantee absence of dependences among explicit threads. Enable thread-private stacks for allocating private variables
	Directive-based postpass	Implicit threads: Control thread selection of the code generator.
f2c	Parser	Parse Fortran Multiplex directives.
Gnu C Code generator	Parser	Parse C Multiplex directives.
	Multiplex backend	Insert special instructions to mark thread execution attributes.
Library	Thread management	Fork and join explicit/implicit threads. Pass arguments from parent to child threads.
	Utilities	Provide thread identification numbers and the number of threads.

Fig. 1. Overview of the compilation system for the Multiplex architecture

analysis, array privatization, idiom recognition, and symbolic analysis [2]. Although the present paper uses automatic parallelization techniques for identifying explicit threads, this is not a prerequisite. For example, explicitly parallel programs (e.g., OpenMP source code) could be used. In such programs, parallel regions would provide candidate explicit threads, while implicit threads would be derived from the "serial" program section.

2.2 Thread Selection

All iterations of parallel loops are candidates for explicit threads. The rest of the program is considered for creating implicit threads. While explicit threading

eliminates unnecessary runtime dependence tracking in parallel code sections, it introduces *explicit threading overhead* due to added instructions and thread management costs.

In Section 4 we describe the heuristic algorithm for selecting explicit and implicit threads. Based on estimated overheads at compile time it may choose to execute fully parallel loop iterations as implicit threads. If insufficient information is available at compile time, then both code versions are generated together with code choosing between the two at runtime.

Our fully-automatic scheme improves on the one presented in [1], in which we have used a semi-automatic method. It also relates to the techniques presented in [15]. In this work, the compiler schedules the outermost loop that will not overflow the speculative storage for thread-level execution, only estimating the usage of the storage. Our scheme is the only existing fully automated technique to consider various characteristics of the applications and to deliver the performance as good as a profiling scheme. The algorithm is a novel part of our compilation system.

2.3 Thread Preprocessing

Thread preprocessing performs two tasks. First, it generates code that makes runtime decisions about whether to execute a thread as implicit or explicit. Second, it transforms explicit loops that are selected for explicit threading into subroutine form.

Two-version Code for Explicit/Implicit Threading If the thread selection algorithm decides that there is insufficient information at compile time, it generates conditional expressions that determine the best threading mode. The preprocessing step inserts an `IF` statement containing this conditional, with the `THEN` and `ELSE` part being the explicit and implicit threading version of the loop, respectively. The explicit threading version selects each iteration of this loop as an explicit thread. The implicit version selects each innermost loop iteration as an implicit thread. Creating this code was straightforward, given the available Polaris infrastructure.

Subroutine Conversion of Explicit Threads Our compiler transforms explicit threads into subroutine form. This transformation is the same as routinely applied by compilers for generating thread-based code from parallel loop-oriented languages. Polaris already contains such a pass, which we have used with small modifications [16]. Figure 2 shows examples of the code transformation in thread preprocessing.

This transformation accomplishes two tasks. First, it provides the explicit thread with a proper stack frame so that a private stack can be used during its execution. Section 2.5 describes the process of setting up stacks for each thread. Second, it guarantees explicit threads to be dependence-free. We have found two causes of false dependences introduced in otherwise fully-independent threads.

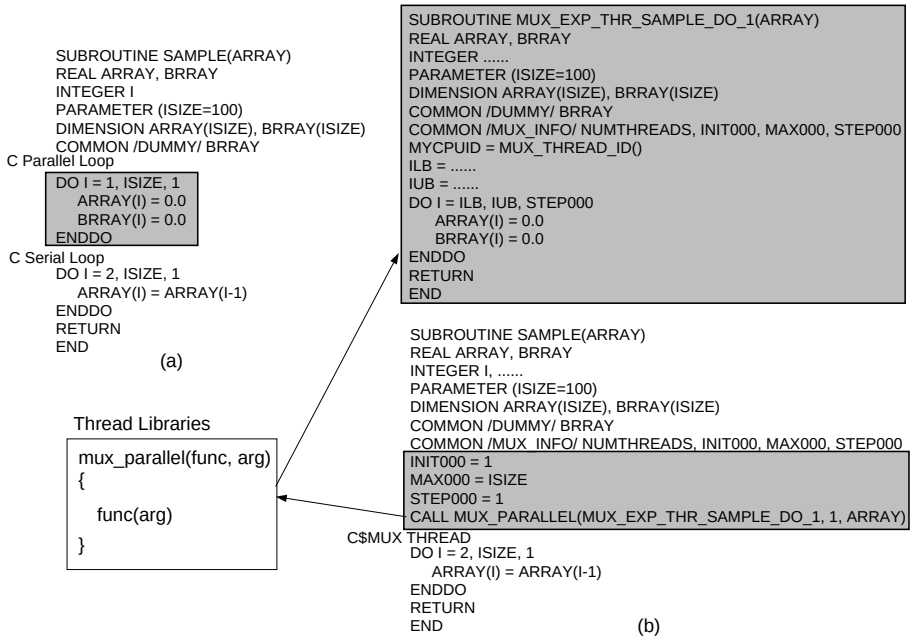


Fig. 2. Code transformation by the preprocessor. (a) Original code. (b) Transformed code. The subroutine `MUX_EXP_THR_SAMPLE_DO_1` is created from a parallel loop, and it is called by the thread manager `mux_parallel` (Section 2.5). The function `mux_parallel` copies shared variables (`ARRAY`) into child threads and control the explicit threads. The newly created subroutines are executed explicitly, while the serial loop is executed implicitly. The serial loop is annotated whith the directive `c$mux thread`.

Although their explanation needs some implementation details, both issues are intrinsic to the architecture and compilation system.

The first type of dependence is introduced as a result of using a state-of-the-art, sequential compiler for low-level code generation. Standard register allocation techniques performed by such compilers attempt to place values in registers. If the lifetime of such a value spans two threads, the two processors executing the threads need to perform *register communication* with proper synchronization. This mechanism is well understood in speculative architectures and is part of our execution scheme for implicit threads. However, for explicit threads such dependences are not allowed. In our compiler we found that the *strength reduction* technique is a major cause of such dependences. The transformation into subroutine form prevents this optimization across explicit threads.

The second type of dependence is only an apparent dependence, in which the processor falsely assumes that register communication is necessary. It is caused by the typical register-save instructions at the beginning and end of a subroutine, which look to the architecture as if the register value were written to memory. In actuality, the value is irrelevant for the subroutine. This problem was iden-

tified in [17] as "artificial dependences" and can cause significant performance degradation in speculative architectures.

2.4 Interfacing with Backend Compiler

The preprocessor analyzes not only parallel loops, but also explicit and implicit threads. This preprocessor information must be correctly passed on to the code generator. The information includes the thread boundaries and attributes (implicit or explicit). It is communicated to the backend code generator in two different forms: Loops that are selected to become implicit threads are annotated with a directive. Explicit threads are identified by a call to a special runtime library and by the name of the subroutine passed as an argument.

A substantial effort in implementing our code generation scheme was necessary to enable the code generator to understand the information obtained through this interface. The issue of interfacing the preprocessor with the code generator is described in more detail in Section 3.

2.5 Code Generation for Implicit and Explicit Threads

The goal of the code generator is to mark implicit and explicit threads in the binary code and insert data necessary for proper threads operation. We use the compiler techniques already provided by the Wisconsin Multiscalar compiler [11]. It inserts thread header information that marks the start of a new thread and expresses cross-thread register dependences. We extended the thread header to mark the thread attributes. We describe the library implementation next.

Thread Libraries The thread libraries provide three functions: a stack manager to initialize the stack space, a thread manager to fork explicit threads, and thread utilities to support the threads' intrinsic functions.

The stack manager initializes a private stack pointer and allocates a private stack space to each processor at the beginning of the program. Private stacks eliminate dependences in thread-private variables of explicit threads.

Figure 3 shows the implementation of the thread manager. It consists of two parts: copying arguments into the private stacks and forking/executing/joining explicit threads.

Two additional thread utility functions are provided: `mux_thread_id` and `mux_num_threads`. The function `mux_thread_id` returns the processor identification number, and `mux_num_threads` returns the number of processors. These functions are used for thread scheduling.

3 Integrating Preprocessor and Code Generator

A substantial effort in implementing our code generation scheme was necessary to build an interface such that the analysis information generated by the preprocessor can be passed on to the code generator. The preprocessor identifies

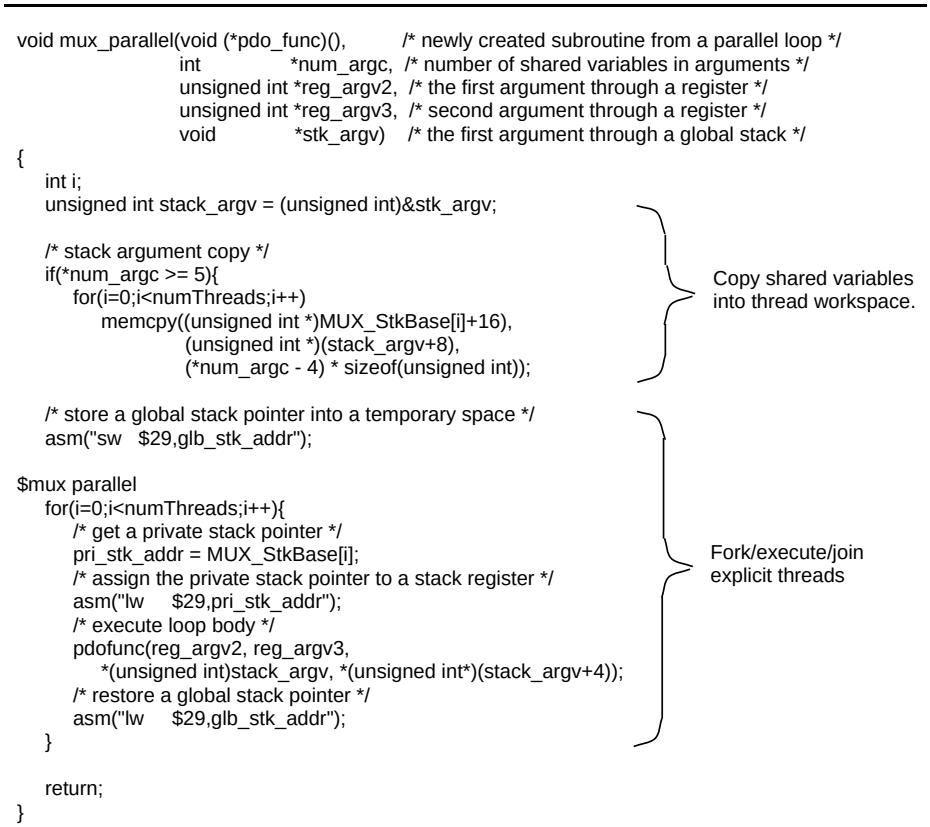


Fig. 3. The explicit thread manager `mux_parallel`. It copies the shared variables into the child thread workspace and controls the explicit threads. The real implementation is done at the assembly level. The program gives an overview of the implementation.

threads and passes their attributes to the code generator. For this purpose, the preprocessor uses a naming scheme for explicit threads. It creates a name prefix of the newly created subroutine for explicit threads, as illustrated in Section 2.3, as `mux_exp_thr_`. The actual code invoking the explicit thread is the call to the `mux_parallel` library routine.

For implicit threads, the preprocessor uses two Multiplex directives: `c$mux thread` to select an iteration of the annotated loop, `c$mux serial_thread` to assign the whole loop to one implicit thread. By default, an iteration of the innermost loop is selected as an implicit thread.

We extended the code generator so that it understands this information. We implemented parsing capabilities for the input directives. We also extended the code generator's internal data structure (RTL, Register Transfer Language) [14] to represent the directive information in the form of *RTL notes*. These notes are

used by the code generating pass that selects implicit threads. For the subroutines of explicit threads the code generator simply produces one-thread code.

Figure 4 shows examples of the implicit thread generation scheme in the original Multiscalar and the new Multiplex compiler. The scheme combines several basic blocks into a thread and marks it as such. The Multiscalar scheme always picks iterations of innermost loops as threads (Figure 4 (b)), whereas the Multiplex compiler picks threads according to the provided `mux$ thread` directives (Figure 4 (c)). Our code generator uses directives from the preprocessor, in the form of extended RTL notes. They describe the boundaries of threads, which can be at any loop level. The basic blocks belonging to the inner loop are copied because the outer loop has two basic blocks 0 and 1 forming the loop entry. The details are described in [12].

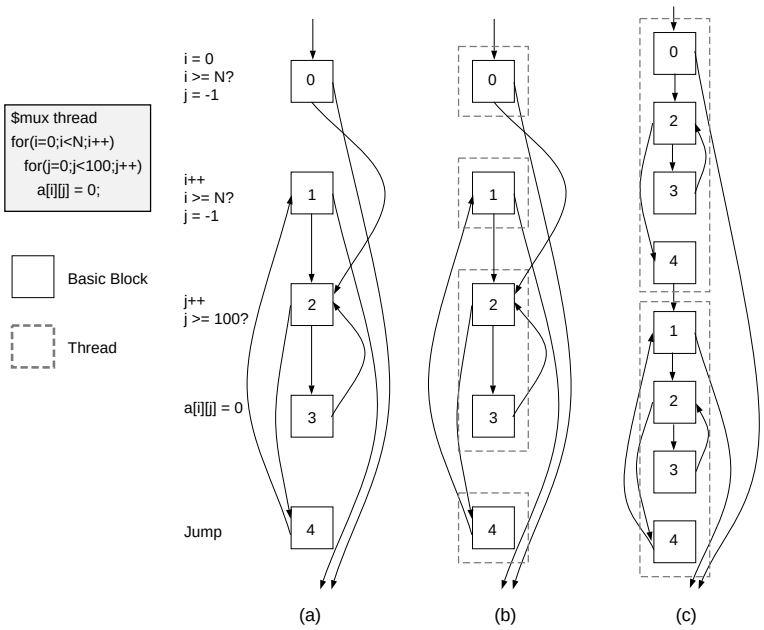


Fig. 4. Thread selection using the Multiplex directive set. (a) Source code and control flow graph (CFG). (b) The Multiscalar compiler. (c) The Multiplex compiler. The thread selection in Multiplex is controlled by the preprocessor, whereas that of Multiscalar is done by the code generator itself, walking through the CFG.

4 Implicit/Explicit Thread Selection

Careful thread selection is important for good performance. A naive scheme may select all iterations of fully parallel loops as explicit threads and all innermost

iterations of the remaining loop nests as implicit threads. Such a scheme would incur overheads that can be reduced with a better selection algorithm. The following factors must be considered.

For our consideration, the most important overhead of implicit threads is *speculative storage overflow*. It occurs when the speculative state (all speculatively written values) built up in the course of the thread execution exceeds the capacity of the speculative storage [18]. Explicit threads do not incur this overhead because they do not use speculative storage. However, they incur overheads due to the added instructions generated by the preprocessing step (subroutine conversion). This overhead is significant in small threads.

Fully parallel code sections can be executed as either implicit or explicit threads. The thread selection algorithm makes this decision by considering the above overheads. Although there are more choices, the algorithm will only consider the outermost parallel loops for explicit execution and the innermost loops for implicit executions. In our experiments we have found that alternative choices can rarely improve performance further. Thus, for each outermost parallel loop $\mathcal{L}$ the algorithm considers the following factors:

Speculative storage overflow: The compiler determines whether the implicit execution of any innermost loop would incur speculative storage overflow. In our architecture, the speculative storage is organized as a direct-mapped L1 cache. Speculative storage overflow occurs on a cache conflict. That is, if the conflicting cache line is currently speculative then it cannot be replaced, hence execution stalls. Our algorithm inspects *read–read* and *write–read* reference pairs in innermost loops of $\mathcal{L}$. If it is able to calculate the address distance of the two references and this distance is a multiple of the number of cache blocks, then a cache conflict is caused by the second reference. The resulting speculative storage overflow can stall the processor execution for several hundred cycles. Therefore, if the compiler detects any such overflow, it chooses explicit thread execution for $\mathcal{L}$. If insufficient information is available at compile time, the compiler marks the loop as needing a dynamic decision and generates the expression that needs to be evaluated at runtime. Note that the algorithm ignores cache conflicts if the second reference is a write. We have found that the store queue, to which write references go, provides sufficient buffering so that the processor does not incur significant stalls.

Loop workload: To estimate $\mathcal{L}$'s workload, the algorithm multiplies the ranges of all indices of enclosing loops. If the result is less than a threshold and there is no speculative storage overflow, then the loop is selected for implicit execution. If the workload is not a constant, then the compiler again marks the loop as needing a runtime decision.

Loop attributes: A candidate loop $\mathcal{L}$ is also executed in explicit mode if any of the following attributes hold. (1) $\mathcal{L}$ is more than doubly-nested (the workload is assumed to be large enough). (2) There is any inner, serial loop, i.e., the loop has compiler-detected cross-iteration dependences (executing inner loop iterations as implicit threads would result in high dependence overhead). (3) $\mathcal{L}$

contains function calls (the algorithm assumes that an implicit execution would incur stalls due to artificial dependences, see Section 2.3).

Algorithm 1 (Selecting Explicit/Implicit Thread Execution Mode) *For each outermost parallel loop $\mathcal{L}$ in the program,*

1. *Estimate the workload of $\mathcal{L}$ and the speculative storage overflow of all innermost loops as described above.*
2. *Select thread execution mode:*
   (a) *Mark $\mathcal{L}$ as explicit if at least one of the following conditions is true:*
      i. *$\mathcal{L}$ is more than doubly nested.*
      ii. *$\mathcal{L}$ encloses any serial loop.*
      iii. *$\mathcal{L}$ contains any function call.*
      iv. *The compiler detects any speculative storage overflow.*
      v. *The workload is greater than the threshold.*
   (b) *Else if both overflow and workload are compile-time decidable, mark all innermost loops of $\mathcal{L}$ as implicit.*
   (c) *Else mark $\mathcal{L}$ as needing runtime decision.*

Figure 5 shows an example of the thread selection decision made in `SWIM INITAL_do70`. The speculative storage is a 4K direct-mapped cache. The algorithm considers only two pairs of accesses: (`u(1, j)`, `v(1, 1+j)`), and (`u(1+m, j)`, `v(1, 1+j)`). At compile time the value of the variable m is not known, so a function call is inserted, `mux_test_overflow`, to test for speculative storage overflow of the two accesses (`u(1+m, j)`, `v(1, 1+j)`) at runtime. The compiler can estimate the overflow of the two variables, `u(1, j)` and `v(1, 1+j)`, because they are in the same common block and the sizes of all variables belonging to the common block are known at compile time. The algorithm also estimates the workload for the loop using the loop index range.

```
      IF (.NOT.mux_test_overflow(u(1+m, 1), v(1, 2)).AND.(-419)+n.LE.0)
     *THEN
C$MUX LOOPLABEL 'INITAL_do70'
C$MUX THREAD
       DO j = 1, n, 1
        u(1, j) = u(1+m, j)
        v(1+m, 1+j) = v(1, 1+j)
   70   CONTINUE
       ENDDO
      ELSE
       init000 = 1
       max000 = n
       step000 = 1
       CALL mux_parallel(mux_exp_thr_inital_do70, 0)
      ENDIF
```

Fig. 5. Code generation for runtime thread selection in `SWIM INITAL_do70`.

The presented scheme is the only existing fully automated technique that considers various characteristics of the application and the hardware to select

the best thread execution mode. In the next section we will show that it performs as well as a scheme that uses profile information.

5 Performance Analysis

Overall Evaluation of the Compiler Infrastructure

Figure 6 shows the speedups of our benchmarks, consisting of the SPECfp95 and three Perfect Benchmarks applications. The bars show (1) the original Multiscalar codes (implicit-only execution, not using our new compiler techniques), (2) our Multiplex execution with a naive thread selection algorithm, (3) the Multiplex execution with our new thread selection algorithm, and (4) the Multiplex execution with a thread-selection scheme based on profile information. The speedups measure the four-CPU execution relative to a one-CPU execution. These results correspond closely to the ones presented in [1], which described the Multiplex hardware system. However, in [1] we used a semi-automatic thread selection scheme, while the measurements in Figure 6 show the performance using our new algorithm. The hardware system has four dual-issue, out-of-order superscalar CPU cores. Each processor has its own private 16K DM L1 data cache as a speculative storage, and the cache coherence protocol is extended to maintain the speculative states. We profiled the performance on a loop-by-loop basis, and manually selected the better execution mode. The Multiplex-naive code executes *all* parallel loops as explicit threads.

For presentation, we classify the measured applications into two: Class 1 (`FPPPP`, `APSI`, `TURB3D`, `APPLU` and `WAVE5`) applications have many small-grain parallel loops inside a serial loop, and Class 2 (the rest of applications) codes have large-grain parallel loops. In the Class 1 applications, the implicit-only execution is better than the naive Multiplex scheme by 6.2% due to the explicit threading overhead. In the other applications (Class 2 applications), the performance in the naive Multiplex codes is better than implicit-only by 43.7%. This improvement is due to the elimination of speculative execution overheads in compiler-identified parallel code sections. Using our compiler infrastructure, the hardware can turn off speculation in compiler-identified parallel code sections. It allows us to exploit large-grain thread-level parallelism, and eliminate the speculative overheads. The results also show that our thread selection algorithm is most effective in the applications of Class 1. We discuss this effect in more detail in the next subsection.

Evaluation of the Thread Selection Algorithm

Figure 6 shows that our heuristic thread selection algorithm improves the naive Multiplex performance in the Class 1 applications by a significant 13.1% on average. The performance of the compiler heuristics does not match that of the profiling scheme in all applications. The main reason is that the compiler does not make a correct choice between implicit and explicit execution because the

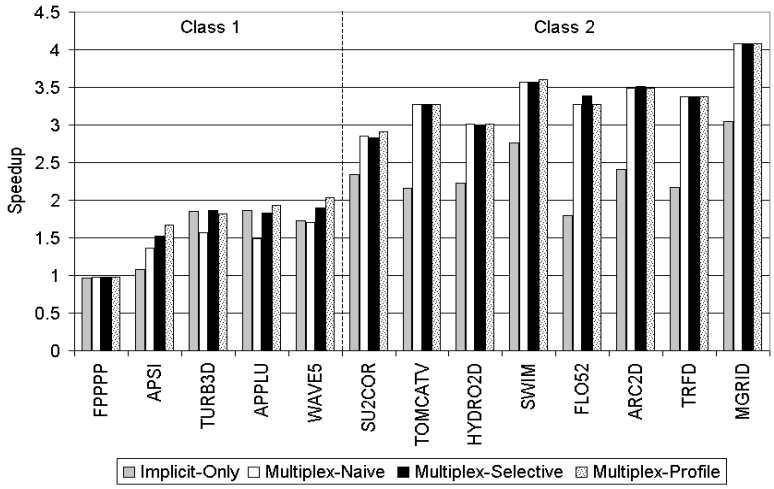


Fig. 6. Overall speedup with respect to the superscalar execution. The "Multiplex-Naive" executes all parallel loops in explicit mode. "Multiplex-Selective" uses our new heuristic algorithm, which executes some of the parallel loops in explicit mode. For comparison, in "Multiplex-Profile" the execution mode is selected manually based on profiling.

array subscripts are not analyzable (not affine). For example, this is the case in the loop `BESPOL_do10` in `SU2COR`.

Figure 7 shows the instruction overhead, which is the ratio of the committed instructions of the four-processor execution schemes to those of a superscalar execution. In the Class 1 applications, the instruction overhead of the naive thread selection algorithm is much higher, resulting in the implicit-only execution performing better. The reason is again the many small explicit loops in those applications, which incur significant explicit threading overhead. The compiler heuristic reduces the instruction overhead in the Class 1 naive Multiplex codes significantly.

6 Conclusions

Speculative architectures have been proposed in order to relieve the compiler from the burden of providing absolute guarantees about program data dependences between sections of code believed to be parallel. However, internally speculative architectures make use of sophisticated techniques that incur intrinsic overheads. Compiler techniques are important to reduce these overheads.

In this paper, we presented the structure of a compilation system that exploits both implicit and explicit thread-level parallelism. The system integrates a parallelizing preprocessor with a code generator for the Multiplex architecture. A preprocessor identifies parallel code sections, and selects the execution

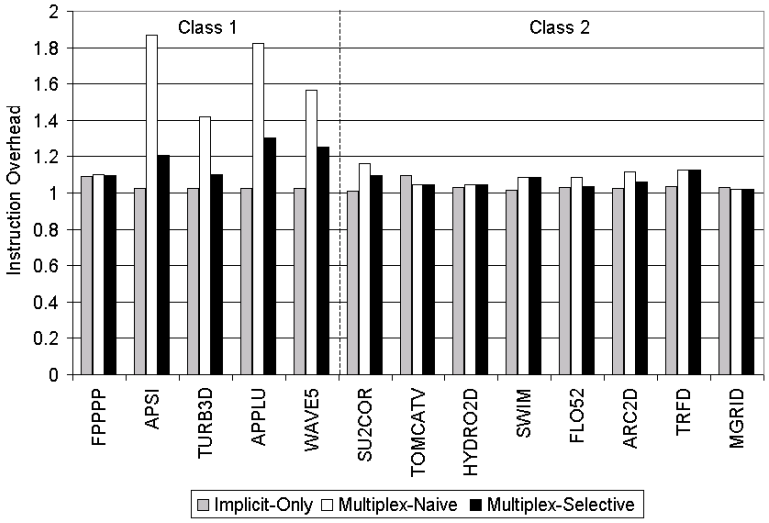


Fig. 7. Instruction overhead of implicit-only, the naive Multiplex and Multiplex using our thread selection algorithm with respect to the superscalar execution. Our algorithm reduces explicit threading overhead significantly in the Class 1 applications.

mode for each section. It also transforms the parallel loops into subroutine-based forms in order to guarantee that there is no dependence for explicit threading. The preprocessor's analysis information (thread boundaries and attributes) is passed to the code generator through an appropriate interface, integrating the two compilers tightly. We also presented a compiler algorithm that decides between implicit and explicit thread execution mode. Where the algorithm cannot make static decisions, it generates runtime decision code. We showed that all of our compiler techniques improve the performance significantly.

There are many related projects in speculative CMPs [6–10]. However, there has not been an emphasis of compilation issues. Our system is the first that can exploit both explicit and implicit thread-level parallelism. Furthermore, our compiler infrastructure is the only existing fully automated translator system for speculative architectures that integrates the preprocessor's analysis capabilities with the code generator.

References

1. Chong-Liang Ooi, Seon Wook Kim, Il Park, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *International Conference on Supercomputing (ICS'01)*, pages 368–380, June 2001.
2. William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger,

Lawrence Rauchwerger, and Peng Tu. Parallel programming with Polaris. *IEEE Computer*, pages 78–82, December 1996.
3. 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.
4. Lawrence Rauchwerger and David Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *The ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI'95)*, pages 218–232, June 1995.
5. Lawrence Rauchwerger and David Padua. The privatizing DOALL test: A run-time technique for DOALL loop identification and array privatization. In *International Conference on Supercomputing (ICS'94)*, pages 33–43, 1994.
6. Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *The 22th International Symposium on Computer Architecture (ISCA-22)*, pages 414–425, June 1995.
7. Kunle Olukotun, Lance Hammond, and Mark Willey. Improving the performance of speculatively parallel applications on the Hydra CMP. In *International Conference on Supercomputing (ICS'99)*, pages 21–30, 1999.
8. J. Gregory Steffan and Todd C. Mowry. The potential for thread-level data speculation in tightly-coupled multiprocessors. Technical Report CSRI-TR-350, University of Toronto, Department of Electrical and Computer Engineering, Feb. 1997.
9. 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.
10. Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In *The Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, pages 162–173, February 1998.
11. T. N. Vijaykumar and Gurindar S. Sohi. Task selection for a multiscalar processor. In *The 31st International Symposium on Microarchitecture (MICRO-31)*, pages 81–92, December 1998.
12. Seon Wook Kim. *Compiler Techniques for Speculative Execution.* PhD thesis, Electrical and Computer Engineering, Purdue University, April 2001.
13. S. I. Feldman, David M. Gay, Mark W. Maimone, and N. L. Schryer. A Fortran-to-C converter. Technical Report Computing Science No. 149, AT&T Bell Laboratories, Murray Hill, NJ, 1995.
14. Richard M. Stallman. *Using and Porting GNU Gcc version 2.7.2*, November 1995.
15. J.-Y. Tsai, Z. Jiang, and P.-C. Yew. Compiler techniques for the superthreaded architectures. *International Journal of Parallel Programming*, 27(1):1–19, February 1999.
16. Seon Wook Kim, Michael Voss, and Rudolf Eigenmann. Performance analysis of parallel compiler backends on shared-memory multiprocessors. In *Compilers for Parallel Computers (CPC2000)*, pages 305–320, January 2000.
17. J. Oplinger, D. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *The 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99), Newport Beach, CA*, pages 303–313, October 1999.
18. Seon Wook Kim, Chong-Liang Ooi, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. Reference idempotency analysis: A framework for optimizing speculative execution. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP01)*, pages 2–11, June 2001.