Integrating Parallelizing Compilation Technology and Processor Architecture for Cost-Effective Concurrent Multithreading

Jenn-Yuan Tsai*, Zhenzhen Jiang*, Zhiyuan Li**, David J. Lilja†, Xin Wang*, Pen-Chung Yew*, Bixia Zheng*, Stephen J. Schwinn†

*Department of Computer Science and Engineering
†Department of Electrical and Computer Engineering
University of Minnesota

**Department of Computer Science
Purdue University

November 26, 1997

Abstract

As the number of transistors on a single chip continues to grow, it is important to think beyond the traditional approaches of compiler optimizations for deeper pipelines and wider instruction issue units to improve performance. This single-threaded execution model limits these approaches to exploiting only the relatively small amount of instruction-level parallelism available in application programs. While integrating an entire multiprocessor onto a single chip is feasible, this architecture is limited to exploiting only relatively coarse-grained parallelism. We propose a concurrent multithreaded architecture, called the superthreaded architecture, as an alternative. As a hybrid of a wide-issue superscalar processor and a multiprocessor-on-a-chip, this new concurrent multithreading architecture can leverage the best of existing and future parallel hardware and compilation technologies. By combining compiler-directed thread-level speculation for control and data dependences with run-time checking of data dependences, the superthreaded architecture can exploit the multiple granularities of parallelism available in general-purpose application programs to reduce the execution time of a single program.

1 Introduction

The past several decades have seen astounding increases in the number of transistors that can be integrated on a single chip. Where a processor used to require several components to implement only the most basic functions, fabrication technology has improved to the point where entire systems can be integrated on one chip. However, since the clock rate in current designs continues to be limited by the interconnect delay between logic components, there is no escaping the fact that additional parallelism must be exploited to continue the expected increases in overall system performance. Factors limiting increases in parallelism today are primarily a lack of innovative software technology and machine architectures that can exploit the available transistors rather than any limitation in hardware technology.

Most of the problems current microprocessor systems have in exploiting parallelism stem from their single-threaded execution model. Consequently, it is natural to consider using multiple threads of execution. With concurrent multiple threads, a processor can exploit instruction-level parallelism from multiple instruction windows simultaneously. The aggregate size of the windows, and the total number of instructions issued in one cycle from all concurrent threads, can be comparable to that of a very large window in a single-threaded architecture, but with much lower hardware cost.

These concurrent multithreaded architectures (CMAs) are similar to having very tightly-coupled multiprocessors on a single chip, yet with very different design goals, architectural supports, and compilation techniques. We propose a new concurrent multithreading architecture, called the *superthreaded* architecture [12], that is a hybrid of current superscalar processors and a multiprocessor-on-a-chip. By tightly integrating the compiler design with the architectural design, we can use the best of the hardware and compilation technologies developed for both superscalar processors and multiprocessors to effectively exploit future transistor densities to reduce the execution time of a single program.

1.1 The Current State-of-the-Art

All existing microprocessors use a superscalar or a VLIW approach to exploit instruction-level parallelism with a single thread of control flow, that is, using only a single program counter. To shorten the clock cycle time, most of them also rely on very deep pipelines. Such architectures require extensive compiler and hardware support to identify and extract instruction-level parallelism from application programs, and to schedule independent instructions on multiple functional units in each machine cycle. To increase the instruction issue rate, the compiler or the hardware must identify independent instructions by analyzing a large instruction window. However, it has been shown to be very difficult and expensive to extract enough parallelism with a single thread of control to efficiently utilize even a small number of functional units [2, 6, 8].

To make matters worse, in most general-purpose applications, there are typically only around 5-10 instructions between two branches, which is the size of a *basic block* [2, 6, 8]. Hence, even without considering data dependences within a basic block, a machine capable of issuing 10 instructions per machine cycle would issue all of the instructions in a typical basic block in only 1 or 2 cycles. Such a machine will encounter a branch instruction every 1 or 2 cycles, on average, and every issue stall caused by a branch instruction will waste 10 instruction slots, which is almost the size of the average basic block.

This situation forces future microprocessor designers to face two fundamental challenges to maintain high issue rates and thereby obtain the desired performance. The first challenge is to find sufficient instruction-level parallelism in user programs to keep a large number of functional units with deep pipelines efficiently utilized. The second challenge is to deal with very frequent branch instructions, potentially multiple branch instructions per machine cycle, to avoid interrupting the deep pipelines.

Existing single-threaded architectures are inadequate to meet these two challenges. For example, a very large instruction window must be used to be able to find enough independent operations to keep the functional units busy. Unfortunately, this large window is very expensive since the hardware necessary for dependence checking among all instructions in the window grows substantially with the size of the window, and any increase in the window size can potentially increase the clock cycle time. Also, a large window will include many basic blocks which may be control dependent on several different branch conditions. To move data independent instructions above branch instructions, instruction-level speculative execution with branch prediction or predicated execution is necessary. However, the accuracy of branch prediction drops quickly after crossing only a few branch instructions, and the profiling information that is often used to select the most frequently executed paths may be input sensitive.

Software pipelining [7] currently is a common technique used to exploit parallelism in loops. In theory, similar techniques could be applied to outer loops as well, but most likely with a substantial loss of efficiency. As a result, most existing software pipelining techniques typically deal only with innermost loops. In addition, the code size may increase exponentially as the level of branch speculation increases since the compiler needs to generate single-threaded code for all possible combinations of branch paths in different loop iterations.

In VLIW architectures, operations in a long instruction word are executed in lock-step to enforce dependences *between* instructions, even though operations in the *same* instruction must be independent to be issued in the same machine cycle. Because of the lock-step execution, any asynchronous event at runtime, such as a cache miss caused by any one of the operations in the instruction word, will stall the execution of all other independent operations in the same instruction. Such unnecessary stalls will happen more frequently, and with more devastating performance effects, as the issue rate (i.e. the width of the instructions) increases.

1.2 Challenges in Parallelizing General-Purpose Applications

Existing microprocessors are the main engines of today's general-purpose computing systems with the targeted application programs exemplified by those selected in the SPEC benchmark suite. While compilation techniques and architectural supports for parallelizing Fortran programs have been studied extensively in the past, it is the non-numerical C and C++ programs that pose the greatest challenges. These types of programs have several characteristics that make it difficult to extract parallelism to produce high performance. These characteristics include:
(1) Most of them have a large number of pointers whose behavior is very difficult to analyze at compile time, thereby limiting the parallelism that can be extracted using traditional compiler-based approaches.
(2) Most of them have loops, such as *do-while* loops, that may terminate as a function of run-time conditions making them essentially impossible to parallelize using standard compile-time techniques.
(3) The most time-consuming inner loop nests in these programs are typically very small, sequential, and full of conditional branches. Potential parallel loops, if they exist at all, are very often in the outer loop nests, and parallelizing on the outer loop has not been adequately addressed by traditional software pipelining techniques.
(4) Most of the programs use procedure calls extensively, many with recursion.

Characteristics (1) and (2) rule out any attempts to put traditional multiprocessors, which are designed primarily for scientific, numerical applications, on to a single chip and expect good performance on these types of programs. Characteristics (3) and (4) further suggest that completely new compiler techniques and architectural supports are necessary to develop a reasonable and sensible approach for extracting parallelism from these irregular applications.

2 The Superthreaded Architecture

The *superthreaded* architecture [12] addresses these challenges in parallelizing general-purpose applications by tightly integrating a parallelizing compiler with a concurrent multithreaded architecture. This tight integration allows the superthreaded architecture to exploit both instruction- and thread-level parallelism using multiple threads of control. In its general form, a superthreaded processor consists of a number of thread processing units with a shared instruction cache and a shared data cache, as shown in Figure 1. At run-time, the multiple thread processing units, each with its own program counter and instruction execution data path, can fetch and execute instructions from multiple program locations simultaneously. Except for the memory buffer, a thread processing unit is very similar to a traditional single-threaded superscalar processor. The addition of the memory buffer allows runtime dependence checking to be performed on potential data dependences identified by the compiler. It also holds the temporary thread context needed for control speculation. Threads are initiated and retired in their original order, and data flows in the same direction as the thread order. Hence, the thread processing units can be connected by a unidirectional wrap-around ring to simplify the hardware complexity. The compiler for the superthreaded architecture must statically partition a program into threads where each thread corresponds to a specific portion of the control flow graph. For instance, a thread could be one or several iterations of a loop.

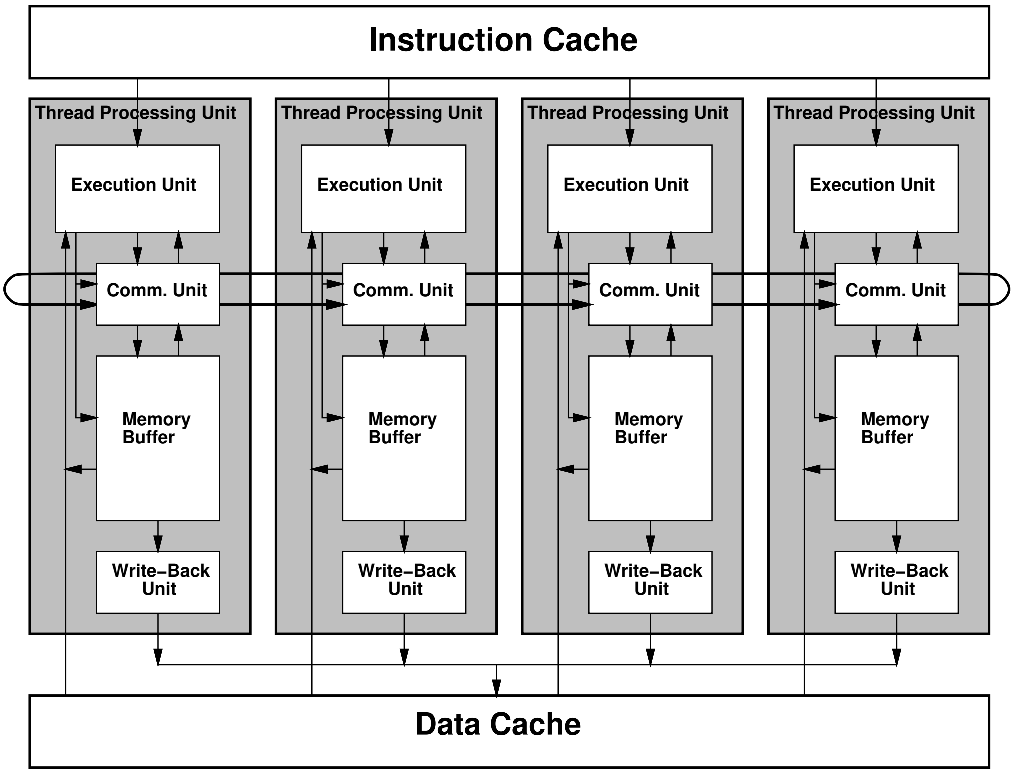

Figure 1: The organization of the superthreaded processor.

As shown in Figure 2, the superthreaded architecture uses a *thread pipelining* model to enforce data dependences between concurrent threads, and to increase the overlap between threads. Each thread execution starts with the continuation stage. In this stage, the recurrence variables and the loop index variables needed to fork the next thread are computed and forwarded. The basic idea is to fork the future threads as soon as possible to increase the overlap between threads. Because the superthreaded architecture does not support speculation on data dependences, a memory buffer is provided in each thread processing unit to store the addresses identified by the compiler as potential data dependences from earlier threads. These addresses, called the *target store addresses*, need to be calculated and forwarded as soon as possible at the target-store-address-generation (TSAG) stage to perform the run time dependence checking.

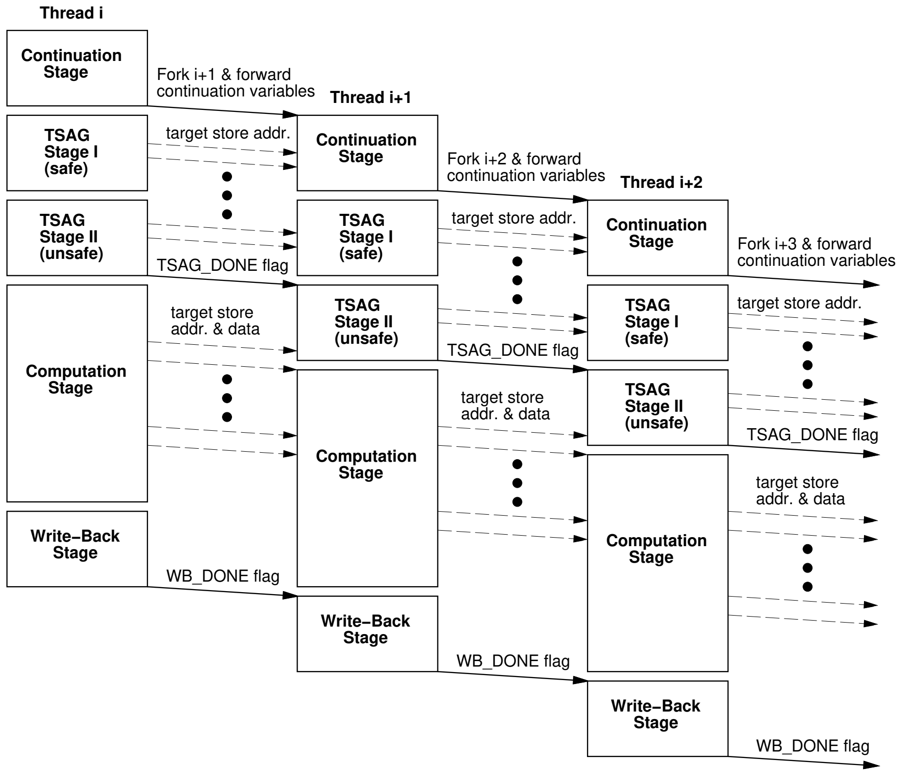

Figure 2: The pipelined execution of contiguous threads.

The computation stage performs the main computation of a thread. If the address of a load operation matches that of a target store entry in its memory buffer, the thread execution unit will either read the data from the entry if it is available, or will wait until the data is forwarded from an earlier concurrent thread. On the other hand, if the value of a target store is computed during this stage, the thread needs to forward the address and the data to the memory buffers of all its concurrent successor threads.

A thread can fork one of its successor threads with or without control speculation. If a thread forks a successor thread with control speculation, it must later verify all of the speculated control dependences. If any of the speculated control dependences are false, the thread must abort all of its successor threads. When all control dependences are cleared, a thread can commit the results of the store operations in its local memory buffer to the main memory. To maintain the correct memory state, concurrent threads perform these write-backs in their original order.

To facilitate thread pipelining and data forwarding, a set of thread management and communication instructions are provided. The thread management instructions, *fork*, *stop* and *abort_future*, are used to implement extremely light-weight threads using only a few instructions to start a new thread. The communication instructions are the target store operations, including: *allocate_ts*, which allocates a target store entry in the memory buffer of the current and successor threads; *store_ts*, which stores the corresponding data in the memory buffer of the current thread and forwards it to the memory buffers of the successor threads; and *release_ts*, which releases the target store entry in the memory buffer. Some simple thread synchronization instructions are also provided.

3 Compilation Issues for Concurrent Multithreaded Architectures

The superthreaded architecture requires more compiler support than is typically provided by traditional front-end parallelizing compilers for multiprocessors since the superthreaded architecture exploits both loop-level and instruction-level parallelism [9]. Existing parallelizing compilers are designed primarily to exploit loop-level parallelism only, and are most appropriate for scientific programs written in Fortran. They usually produce annotated source programs and rely on back-end compilers to perform instruction level optimizations. Currently, these two processes are rarely integrated. Hence, the optimizations performed in the front-end parallelizing compiler are often ignored, or even get "re-optimized" away, by a back-end optimizing compiler which has its own, totally different, optimization targets. It is not unusual to observe lower performance after the program is "optimized" by both the front-end and the back-end than when optimizations are turned off on today's superscalar-based multiprocessor workstations. However, it is very time consuming to build a new integrated compiler and bring both its parallelizing compiler and back-end compiler up to the state-of-the-art.

We have developed a universal *high-level interface* (HLI) which allows us to leverage existing compiler infrastructures in an integrated approach. It is based on the observation that the internal data structures required in front-end parallelizing compilers are different from those needed in the back-end compilers because of the nature of their analyses and program transformations. For example, a parallelizing compiler does not need detailed low-level, machine-dependent information as does the back-end compiler. Carrying less information in the front-end compiler can save both the memory space and the time to maintain it. Integration then requires only a mechanism to transfer the needed information from the front-end compiler to the back-end compiler – it does not require them to have uniform internal data structures (even though that ultimately may be the most desirable approach).

The high-level program information needed in the back-end compiler, which can be provided by the front-end parallelizing compiler, includes loop-carried data dependences, aliasing information, results of interprocedural analysis, and loop parallelization information. The HLI can be easily extracted from a parallelizing front-end compiler and imported by a back-end compiler, as shown in Figure 3. This front-end/back-end compiler interface allows the following compilation issues to be addressed for the superthreaded architecture.

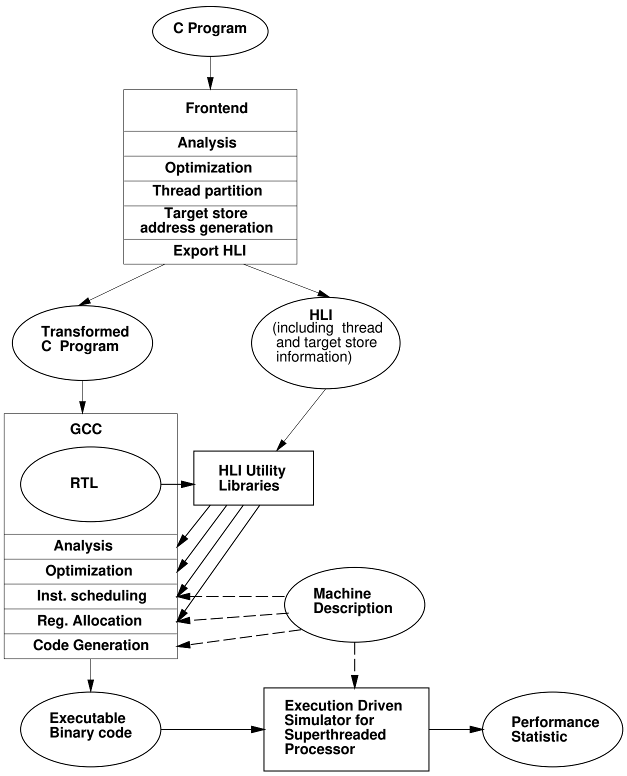

Figure 3: The High-Level Interface (HLI) between the parallelizing front-end compiler and gcc.

3.1 Thread-Pipelining Code Generation

To partition a computation into the four thread pipelining stages, the compiler must identify both loop recurrence variables and *maybe* and *definite* data dependences. Accurate and precise identification can increase the overlap between threads and can minimize the required synchronization. Because the hardware supports thread-level speculation, *dowhile* loops can be executed as parallel *doacross* loops (i.e. loops with cross-iteration dependences). This execution mode is particularly useful for C programs where *dowhile* loops are common. To fully exploit this mode, the compiler must convert all *dowhile* loops into regular *do* loops with explicit exit statements. Additionally, arrays referenced using pointers must be converted into subscripted arrays.

These conversions allow many traditional parallelization and analysis techniques to be applied directly to these loops. These conversions also introduce many new compilation challenges, such as the need for more advanced induction variable recognition and elimination techniques, more detailed flow analysis, and so on [1]. Program transformation techniques that can improve the overlap between loop iterations can be applied here with additional consideration for thread pipelining. Knowing the extent of thread overlap could help determine whether a loop should be speculated. If the overlap is too small, for instance, the chance of a thread being squashed will be too high to justify its speculative execution.

3.2 Thread Selection Strategies

The criteria necessary to identify the proper loop nest for thread pipelining are important for appropriately applying loop interchanging techniques. These criteria are similar to those used in multiprocessors, but with additional constraints. For instance, the compiler must select a loop nest that will provide as large a granularity in each thread as possible while exploiting the maximum amount of parallelism among threads.

The superthreaded architecture has more flexibility in this regard than multiprocessors since it allows *do-while* loops (i.e. those with an indeterminate number of iterations) to be considered for parallelization during the loop selection phase, in addition to loops with a fixed number of iterations. Thread selection should also minimize the amount of communication between threads, and the amount of context that needs to be stored for thread speculation. Too large a context will cause the memory buffer to overflow, which will stall the thread execution.

3.3 Compile-time Analysis and Optimization

To allow better parallelization of loops in C programs, interprocedural pointer analysis is needed for both scalar and array pointers, including static and dynamic alias analysis. So far, most of the work in this area has focused on scalar pointers only. This analysis can be extended to include array pointers inside of loops to allow more loops to be parallelized. With the runtime data dependence support in the superthreaded architecture, some of the aliases can be detected at runtime, which allows for a more relaxed compile-time analysis and more flexible code optimizations.

3.4 Memory Latency and Locality Issues

Since all of the threads in a superthreaded processor will access the same data and instruction caches, traditional memory optimizations, such as locality enhancement through loop tiling and loop unrolling, and latency hiding strategies, such as data prefetching [14], can be applied. However, these compiler optimization techniques must consider the unique characteristics of the superthreaded architecture. For example, with multiple threads sharing the same cache memory, there may be more conflict misses requiring a more careful data layout by the compiler.

Furthermore, because there will be no need for cache coherence enforcement, there will be no false-sharing problems. In fact, sharing of the cache lines among threads should be encouraged to improve locality. Prefetching one iteration ahead [10] will not work properly since several iterations will be active concurrently. Also, because of the shared cache memory, a cache block prefetched by one thread can be used by all other threads as well. As a result, many redundant prefetching instructions may be eliminated.

3.5 An Example Program

The code segment shown in Figure 4 is one of the most time-consuming loops in the SPECint95 benchmark *124.m88ksim*. This loop is a *while* loop that has exit conditions in both the loop head and the loop body. It also has a potential read-after-write data dependence across loop iterations caused by the variable `minclk`.

Figure 5 shows the equivalent superthreaded code for this loop. In this code, each thread corresponds to a loop iteration. In the continuation stage, each thread increments the recurrence variable `i` and forwards its new value to the next thread processing unit using a *store_ts* instruction. The original value of `i` is saved in `i_1` for later use. The continuation stage ends with a *fork* instruction to initiate the successor thread.

In each thread, there is only one target store corresponding to the update of the variable `minclk`. The address of the variable `minclk` is forwarded to the next thread in the TSAG stage. Since the TSAG stage is not dependent on any predecessor threads, it can proceed immediately after the continuation stage. However, the computation stage must wait until the predecessor threads complete the TSAG stage and forward all of the target store addresses. This synchronization is enforced with the *tsag_done* flag between threads.

In the computation stage, a thread begins by checking if the first exit condition is true. If it is, the thread aborts the successor threads with the *abort_future* instruction, and then jumps out of the loop. Otherwise, the thread will perform the computation of the loop body. In the computation stage, the update to the variable `minclk` is performed using a *store_ts* instruction, which forwards the result to its successor threads. If the control path that executes the *store_ts* is not taken, the thread will execute a *release_ts* instruction to release the target store entry so that the successor threads will not wait for the target store data. If both exit conditions are false, the thread will execute a *stop* instruction and will begin the write-back, which is performed automatically by the hardware.

```
while ( funct_units[i].class != ILLEGAL_CLASS ) {
    if( f->class == funct_units[i].class ) {
        if ( minclk > funct_units[i].busy ) {
            minclk = funct_units[i].busy;
            j = i;
            if ( minclk == 0 ) break;
        }
    }
    i++;
}
```

Figure 4: The original example source code from *124.m88ksim*

4 Preliminary Performance Evaluation

We have developed a simulator for a superthreaded processor to make some preliminary comparisons of its performance to a traditional single-threaded processor [12]. The superthreaded processor consisted of either 4 or 8 thread processing units where each unit can issue a single instruction per cycle, has in-order instruction execution, can forward one target store entry per cycle, and assumes a one-cycle delay for all loads and branches. The single-threaded processor is a conventional RISC processor capable of issuing one instruction per cycle with in-order instruction execution and a one-cycle delay for both loads and branches. The number of useful instructions executed per cycle (IPC) by the superthreaded processor when executing several of the SPECint95 benchmarks and GNU text utilities are shown in Figure 6. This table also shows the speedup of the superthreaded processor relative to the single-threaded processor.

5 Alternative Architectures

The similarity between the superthreaded architecture and existing small-scale shared-memory multiprocessors composed of superscalar processors allows the superthreaded architecture to leverage existing research in parallel hardware and compiler technologies while providing extra features to avoid many of the shortcomings that prevent existing multiprocessors from becoming tomorrow's general-purpose computing engines. One key difference of the superthreaded architecture compared to a multiprocessor is that the superthreaded architecture supports thread-level control speculation with runtime data dependence checking. Another important difference is that communication between threads in the superthreaded architecture does not need to go through the memory, which is necessary in a multiprocessor. This difference reduces the communication overhead of the superthreaded architecture allowing it to exploit lighter-weight threads (i.e. finer-grained parallelism) than is possible in a multiprocessor. Also, since the threads are more tightly coupled than in a multiprocessor, the superthreaded architecture requires a less complicated interconnection network, and, since the threads share the same data cache, the cache coherence problem is eliminated. This tighter coupling allows for more efficient execution of general-purpose application programs written in languages such as C and C++, in addition to numerical scientific applications written in Fortran, compared to traditional shared-memory multiprocessor systems.

To increase the instruction issue rate in a "single-processor" architecture, several other multithreaded architectures have also been proposed, including the *XIMD* [15], the *Elementary Multithreading* architecture [5], the *M-machine* [4], the *Simultaneous Multithreading* architecture [13], the *Multiscalar* [11], and the *SPSM* [3]. The key features of the different concurrent multithreaded architectures are compared in Figure 7. The *Simultaneous Multithreading* and *SPSM* architectures support only independent parallel threads, such as parallel loops without cross-iteration dependences (i.e. *do-all* loops), while architectures such as the *XIMD*, the *Elementary Multithreading* architecture, the *M-machine*, and the *Multiscalar* can parallelize loops with cross-iteration dependences using data synchronization and communication between threads.

To enhance the thread-level parallelism that can be exploited, both the *SPSM* and the *Multiscalar* support speculation on both control and data dependences between threads. With support for control speculation, a thread that is control dependent on conditions determined by earlier threads can be initiated and speculatively executed before the results of the condition are known. After the conditions are determined, a speculative thread is either committed or squashed. Such thread-level control speculation allows a *do-while* loop to be executed as a parallel loop. Data speculation support, on the other hand, allows a thread to speculatively assume *no dependence* between operations and to execute load instructions before potentially dependent store instructions from earlier threads are executed. If a data dependence violation is detected at run-time, the thread that violates the data dependence must be squashed. Hardware support for data speculation can be very expensive, however, since all load and store addresses from all active threads must be saved for run-time detection of data dependence violations. The *Multiscalar* uses the global *Address Resolution Buffer* (ARB) for this purpose.

The superthreaded architecture also provides hardware support for thread-level control speculation. However, it does not *speculate* on data dependences, but instead performs runtime data dependence *checks*. As a result, it needs to keep only the *store* addresses in each thread processing unit's memory buffer, but no *load* addresses. Since there are usually far more *loads* than *stores* in most application programs, not needing to save *load* addresses allows the size of the memory buffers in the superthreaded architecture to be much smaller than in the *Multiscalar*.

Among these architectures, only the *Elementary Multithreading* and the *M-machine* use multiple threads to both hide memory latency and increase the instruction issue rate. In the *Elementary Multithreading* machine, a processor provides a number of thread slots and register sets to accommodate multiple threads. At run-time only a few threads can be active and assigned to the thread slots for execution. When an active thread executed on a thread slot encounters a long-latency memory access, the processor will assign the thread slot to another ready thread residing in a register set. This approach can increase instruction issue rate as well as hide memory latency. In the *M-machine*, a processor contains four execution clusters and can support up to six resident *V-threads*. A *V-thread* is similar to a standard process and is composed of up to four *H-threads* that are concurrently executed on the four execution clusters. Each cluster context switches between *H-threads* of different *V-threads* for hiding memory latency. Although it is not described in this paper, it is possible to extend the superthreaded architecture to use multiple threads to hide memory latency as well as to increase the instruction issue rate.

6 Conclusions

As the number of transistors that can be integrated on a single chip continues to grow, it is imperative that computer architects think beyond the traditional approaches for improving performance. Interconnect delays will continue to limit the clock frequency, leaving parallelism as the most promising alternative. However, the single-threaded execution model of VLIW and superscalar architectures limits the amount of parallelism they can exploit, while multiprocessors-on-a-chip can exploit primarily coarse-grained loop-level parallelism. The superthreaded architecture, on the other hand, is a hybrid of both superscalar and multiprocessor architectures allowing it to leverage the best of existing hardware and compilation technologies. This concurrent multithreaded architecture exploits parallelism at both the instruction level and the thread level using run-time data dependence checking with speculation of control dependences. Exploiting these multiple levels of parallelism is made possible only by tightly integrating the parallelizing compiler with the superthreaded architecture.

Acknowledgements

This work was supported in part by the U.S. Army Intelligence Center and Fort Huachuca under contract DABT63-95-C-0127 and ARPA order no. D 346, and by the National Science Foundation under grants MIP-9610379, MIP-9496320, and CDA-9502979. Additional support was provided by a gift from the Intel Corporation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

References

[1] Randy Allen and Steve Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 241–249, June 22–24, 1988.

[2] Michael Butler, Tse-Yu Yeh, Yale Patt, Mitch Alsup, Hunter Scales, and Michael Shebanow. Single instruction stream parallelism is greater than two. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 276–286, May 27–30, 1991.

[3] Pradeep K. Dubey, Kevin O'Brien, Kathryn O'Brien, and Charles Barton. Single-program speculative multithreading (SPSM) architecture: Compiler-assisted fine-grained multithreading. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 109–121, June 27–29, 1995.

[4] Marco Fillo, Stephen W. Keckler, Dally William J, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 146–156, November 29–December 1, 1995.

[5] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, and Teiji Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 136–145, May 19–21, 1992.

[6] Norman P. Jouppi and David W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272–282, April 3–6, 1989.

[7] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, June 22–24, 1988.

[8] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, May 19–21, 1992.

[9] Zhiyuan Li, Jenn-Yuan Tsai, Xin Wang, Pen-Chung Yew, and Bixia Zheng. Compiler techniques for concurrent multithreading with hardware speculation support. In *Proceedings of the 9th Workshop on Languages and Compilers for Parallel Computing*, August 8–10, 1996.

[10] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural*

Support for Programming Languages and Operating Systems, pages 62–73, October 12–15, 1992.

[11] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 22–24, 1995.

[12] Jenn-Yuan Tsai and Pen-Chung Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques, PACT '96*, pages 35–46, October 20–23, 1996.

[13] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 22–24, 1995.

[14] Steven VanderWiel and David J. Lilja. When caches are not enough: Data prefetching techniques. In *IEEE Computer*, volume 30, pages 23–30, July 1997.

[15] Andrew Wolfe and John P. Shen. A variable instruction stream extension to the VLIW architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–14, April 8–11, 1991.

```
/* Continuation Stage */
L1:
    i_1 = i;
    store_ts(&i,i_1+1);
    fork L1;

/* Target-Store-Address-Generation Stage */
    allocate_ts(&minclk);
    wait_tsag_done;
    release_tsag_done;

/* Computation Stage */
    if (funct_units[i_1].class == ILLEGAL_CLASS ) {
        abort_future;
        i = i_1;
        goto L2;
    }

    if ( f->class == funct_units[i_1].class ) {
        if ( minclk > funct_units[i_1].busy ) {
            store_ts(&minclk, funct_units[i_1].busy);
            j = i_1;

            /* if minclk is zero, break to terminate search */
            if ( minclk == 0 ) {
                abort_future;
                i = i_1;
                goto L2;
            }
        } else
            release_ts(&minclk);
    } else
        release_ts(&minclk);

    stop;
/* Write-back Stage */
/* -> performed automatically after stop */
/* End of thread pipelining */

L2:  /* continue */
```

Figure 5: The equivalent superthreaded code for the example program in Figure 4.

Benchmark	% of program simulated	IPC		Speedup		Subroutines simulated
		4	8	4	8	
ear	98.6	3.59	5.38	3.53	5.29	EARSTEP, sos, agc, hwr, difference, fos
m88ksim	75.8	2.64	3.34	2.72	3.53	killtime, ckbrkpts, alignd, test_issue, test_issue, loadmem, rdwr
gcc	29.8	1.22	1.34	1.37	1.50	cse_main, regclass
compress	90.2	1.13	1.13	1.13	1.13	compress, decompress
li	49.0	1.05	1.05	0.96	0.96	mark, sweep, xlygetvalue, xlxgetvalue
ijpeg	69.9	3.08	4.60	2.91	4.35	jpeg_fdct_islow, jpeg_idct_islow, fullsize_smooth_downsample, forward_DCT, rgb_ycc_convert, h2v2_smooth_downsample, h2v2_merged_upsample
wc	99.7	2.76	3.83	1.95	2.71	wc
cmp	99.8	3.20	5.90	3.34	6.15	block_compare_and_count
alvinn	88.6	3.70	7.36	3.61	7.17	input_hidden, hidden_output, output_hidden, hidden_input, update_stats update_weights

Figure 6: Performance summary of the superthreaded architecture with 4 and 8 thread execution units showing the number of useful instructions issued per cycle (IPC) and its speedup compared to a single-threaded processor.

	XIMD	M-machine	Multiscalar	SPSM	Superthread
Control speculation	no	no	yes	yes	yes
Data speculation	no	no	yes	yes	no
Memory buffering	no	no	load/store	load/store	store
Run-time data dependence checking	no	no	yes (squash/ enforce)	yes (squash)	yes (enforce)
Data transfer between threads	register/ memory	register/ memory	register/ memory	no	memory
Data synch	barrier	full/empty bit	register mask	no	target store

Figure 7: Comparison of the key features of concurrent multithreaded architectures.