When All Else Fails, Guess: The Use of Speculative Multithreading for High-Performance Computing

Technical Report No: ARCTiC-00-06 June 2000

David J. Lilja

UNIVERSITY OF MINNESOTA
Laboratory for Advanced Research in Computing Technology and Compilers
Department of Electrical and Computer Engineering • 200 Union St. SE • Minneapolis • Minnesota • 55455 • USA

When All Else Fails, Guess: The Use of Speculative Multithreading for High-Performance Computing

David J. Lilja*

Abstract— **Fundamental physical limits are being encountered in the design of integrated circuits that will limit future increases in processor clock rates. As a result, computer architects are developing aggressive new mechanisms to execute instructions speculatively, that is, before it is known whether or not they should actually be executed, and even before the input values needed by the instructions have been computed. Our speculative multithreading execution model combines compiler-directed thread-level speculation of control dependences with run-time verification of data dependences. This new approach attempts to exploit the fine-grained parallelism available in program constructs that have been resistant to traditional parallelization techniques, such as do-while loops, pointers, and subscripted subscripts. This speculative multithreading model is supported by our superthreaded processor architecture, which is a hybrid of a wide-issue superscalar processor and a multiprocessor-on-a-chip. We also have developed libraries for both C and Java to extend this speculative multithreading to off-the-shelf shared-memory multiprocessors. This paper summarizes our previously reported work on the superthreaded architecture and provides an update on its current status and performance.**

Keywords— **superthreading, multithreading, speculation, dependences, computer architecture, computer design**

I. Introduction

Numerically-intensive application programs, such as those from scientific and engineering fields, significantly stress a computer system's ability to access, process and store large amounts of data. As a result, these applications often have been seen as the ultimate test of a computer system's performance capability. A consequence of this viewpoint is that much of the computer architecture research to date has focused on improving the performance of computer systems for these types of applications, even though they represent only about 15% of the overall market for large systems [1]. An implicit assumption in this previous research has been that *all* types of computer systems would benefit from the architectural innovations developed to improve the performance of this subset of applications. However, nonnumeric application programs, specifically, those that do not operate on regularly structured data stored in large arrays, cannot benefit directly from these previous architectural approaches.

Application programs for nonnumeric processing, such as on-line transaction processing, file-serving, data-mining, and web-serving, for instance, can be as computationally challenging as numerically-intensive applications, yet share few of their processing characteristics [2], [3], [4], [5], [6]. While numerically-intensive applications tend to iteratively apply a fixed number of operations to large sets of regularly structured data, the more general nonnumeric applications tend to apply a range of different operations to each of several widely dispersed data elements. This behavior requires complex control flows that lead to more frequent, and less easily predicted, branch instructions, which degrades the effectiveness of branch prediction hardware. Furthermore, these complex control structures yield less dense code and larger instruction cache footprints, which then leads to more instruction cache capacity misses. Similarly, data cache performance is often degraded by the poor spatial locality associated with processing irregularly structured data. Since nonnumeric applications often operate upon data structures that are held together via pointers, these applications often cannot take advantage of techniques such as cache blocking [7], [8] or data prefetching [9] that rely on tightly packed, regularly spaced data structures.

Taken together, these characteristics of nonnumeric applications cause existing architectural techniques that rely on repeating the same set of operations over many different data elements to be relatively less effective when used to improve the performance of these more general applications. This paper summarizes our previously reported work on the superthreaded architecture, which attempts to improve the performance of these types of less regular application programs without degrading the performance that can be obtained on numerically-intensive applications. In particular, we describe our speculative multithreading execution model, which combines compiler-directed thread-level speculation of control dependences with run-time verification of data dependences. This execution model can exploit the fine-grained parallelism available in program constructs that have been resistant to traditional parallelization techniques, such as `while` loops, pointers, and subscripted subscripts. The superthreaded processor architecture that we have developed to support this multithreading execution model is then described, along with software libraries that we have developed for both the C and Java programming languages to extend this speculative multithreading model to off-the-shelf shared-memory multiprocessor systems.

II. The Problem

A. Performance Trade-offs

The fundamental measure of performance in which we are interested is the time required to execute an application

*Department of Electrical and Computer Engineering, and Minnesota Supercomputing Institute, University of Minnesota, 200 Union St. SE, Minneapolis, MN 55455, E-mail: lilja@ece.umn.edu

program [10]. A simple analytical model for this execution time is:

$$T_e = n \times CPI \times T_{clock}, \quad (1)$$

where T_e is the total execution time, n is the number of instructions executed, CPI is the average number of clock cycles required to execute each instruction, and T_{clock} is the period of the processor's clock. While this model ignores some important aspects of the system that can substantially influence performance, such as the memory and input/output subsystems, it does emphasize several of the primary performance factors that a processor designer can control.

Perhaps the most direct approach to improve performance is simply to reduce the processor's cycle time, T_{clock}, by scaling a given processor design to a faster technology. This approach can often lead to dramatic improvements in performance. However, not all of the signal propagation delays on a chip scale down at the same rate for any particular technology. Furthermore, there are fundamental physical limits that will constrain this straight-forward approach for improving performance.

Another approach to improve performance is to make each individual instruction in the processor's instruction set architecture (ISA) do more work per cycle. Assuming that the total amount of work that must be completed by the application program is fixed, fewer instructions will need to be executed. This change corresponds to reducing the factor n in Equation 1. However, changing the work performed by each instruction will most likely make it more complex to implement, requiring more levels of logic, for instance. This increase in complexity will necessitate a corresponding increase in the processor's cycle time, T_{clock}. Since there typically is not a proportional relationship between these two factors, decreasing n by altering the ISA may not necessarily improve the overall performance. In fact, this trade-off is at the heart of the reduced instruction set (RISC) versus complex instruction set (CISC) debate [11].

The single remaining factor over which a processor designer has control is the average number of clock cycles required to execute each instruction, CPI. Reducing the number of clocks required to execute an instruction typically requires more work to be performed in each clock period. As in changing the factor n above, however, this change will most likely cause an increase in T_{clock}.

An alternative approach to reducing CPI is not to change each individual instruction, but, instead, to increase the number of instructions that are executing simultaneously. While each instruction still requires the same number of clock cycles to execute, the average number of cycles required per instruction will be reduced. When enough instructions are executed in parallel so that the $CPI < 1$, it is helpful to rewrite Equation 1 in the following form:

$$T_e = n \times T_{clock}/IPC, \quad (2)$$

where $IPC = 1/CPI$. That is, IPC is the average number of Instructions issued Per Cycle. Since the IPC can typically be increased faster than the increases in T_{clock} that may be required to support this change, executing more instructions in parallel is a very promising approach to improve performance.

B. Parallelization Challenges

Numerous compilation techniques and parallel architectures have been developed that have been used to successfully parallelize regularly-structured scientific and numerical application programs, such as those commonly written in Fortran [1]. However, parallelizing less regular applications written in languages such as C, C++, and Java has proven to be much more difficult due to several characteristics that occur frequently in programs written in these languages. In particular, these languages encourage the use of pointers, which are difficult, if not impossible, to analyze completely at compile-time. As a result, the compiler cannot determine if dependences exist between program statements. It must then make the conservative assumption that the dependences will exist at run-time and so must sequentialize those statements.

Another program construct that makes compile-time parallelization difficult is the use of `while` loops that terminate as a function of results computed at run-time. Unfortunately, the number of iterations that will be executed by a loop must be known at compile-time to be able to parallelize the loop using conventional techniques. Consequently, these types of run-time dependent loops also must be sequentialized.

Another factor that must be considered is that parallelizing a loop introduces some overhead. As a result, a sufficiently large number of iterations of the loop must be executed to compensate for the overhead introduced by the parallelization. Obviously, the most-timing consuming loops are the best candidates for parallelization since speeding-up these loops will have the greatest impact on overall performance. However, these loops, which are typically the inner loop of a nest of loops, often execute a relatively small number of iterations. Furthermore, it is not unusual for the body of the loop to contain complex branching conditions and relatively few executable statements.

As an example of these complex program structures, Figure 1 shows one of the most important loops in the *m88ksim* program from the SPECint95 benchmark suite. The number of iterations that will be executed by this `while` loop are unknown at compile-time. Furthermore, it has two potential exit points, one at the head of the loop and one at the `break` statement. The body of the loop consists primarily of nested conditional statements. Finally, the variable `minclk` introduces a potential dependence between iterations.

Taken together, these program characteristics make it very difficult to parallelize these types of programs using conventional techniques. In the next section, we introduce our speculative multithreading execution model that can be used to effectively parallelize these traditionally hard-to-parallelize application programs.

```
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++;
}
```

Fig. 1. An example loop from the *m88ksim* program from the SPEC95 benchmark suite.

III. Speculative Multithreading

A. Maybe Dependences

The key to the speculative multithreading execution model is that it allows program optimization decisions to be deferred as long as possible. The compiler then can make the most aggressive assumptions possible about program characteristics to allow optimizations, such as parallelization, to be performed that would otherwise be impossible. The compiler then relies on the hardware to detect at run-time whether or not the correct assumption was made at compile-time. If the behavior at run-time turns out to be different than the assumption the compiler made, the hardware must take appropriate action.

For instance, *maybe* dependences are those dependences detected by the compiler for which it cannot conclusively determine whether or not the dependence will actually exist at run-time. In a conventional system, the compiler must conservatively assume that all *maybe* dependences will in fact occur at run-time. It then must sequentialize the corresponding program statements to ensure that no dependences will be violated at run-time.

In our speculative parallelization model, however, the compiler can assume that *maybe* dependences will not actually occur at run-time. The compiler can now schedule the corresponding statements to execute concurrently while relying on special hardware to detect if the dependence does actually occur when the program is executed. If it does, the hardware must ensure that no changes are made to the global system state as a result of the compiler's incorrect assumption. Given appropriate hardware support, this parallel execution model allows the system to speculate on control dependences and to perform run-time checks on data dependences.

B. Programming Model

The speculative multithreading programming model assumes a system with p independent processors that will act as the processing units for each thread. The compiler must partition a loop into the threads that will be executed on these separate thread processing units. Typically, each thread will consist of the body of the loop that executes a single iteration. Each thread is then subdivided into the separate stages shown in Figure 2. The *continuation* stage consists of those instructions that are necessary to calculate the values needed to start the next thread. Typically, this stage consists of nothing more than a simple increment of the loop iteration counter. This stage ends with a `fork` instruction that causes the next thread to begin executing on the next available thread processing unit. This new thread then computes the continuation values needed to start the next thread, and executes a `fork` to start the next thread. This process continues until all of the available thread units are busy.

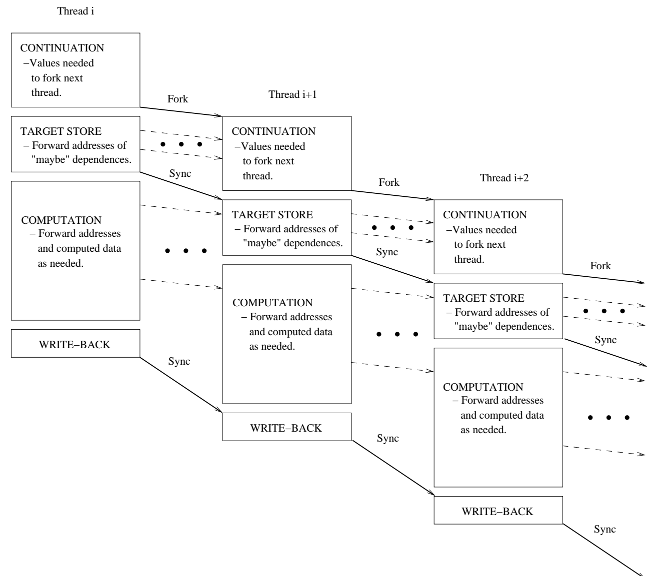

Fig. 2. The speculative multithreading execution model.

One important point to note in this thread initiation process is that only the first thread that executes, which is called the *head* thread, is nonspeculative. All of the threads that are subsequently initiated as children of this head thread are executed in a speculative mode. This mode means that any values that these threads produce must be stored in a private local memory. A thread's speculative values cannot be written back to the global memory until the thread becomes the nonspeculative head thread. This passing of the nonspeculative head state to subsequent thread processing units is described further below.

The second phase of a thread's execution is the *target store address generation* stage. In this stage, the thread computes the addresses of all variables that the compiler previously identified as being a participant in a *maybe* dependence. These computed addresses then are forwarded to the child thread that was forked at the end of this thread's continuation stage. These target addresses must be stored in a special buffer within the child thread. They are subsequently used to determine whether the corresponding *maybe* dependence actually occurred or not. This stage ends with a synchronization instruction that allows the child thread to continue into its target store address generation stage. The parent thread simultaneously continues into its computation stage without needing to wait for a response from its child thread.

The real work contained within the body of a thread is performed in the *computation* stage. The key to the run-time dependence checking is that, when a thread ex-

ecutes a load instruction in this stage, it checks the address of the load to determine if it matches any of the target store addresses previously forwarded from its parent thread. A matching address indicates that the value for the load instruction will be (or has been) produced by the parent thread. In this case, the thread must use the value forwarded to it from its parent. If the value is not yet available, it pauses its execution until the value is passed from its parent. If the address being referenced by the load does not match one of the target store addresses, however, the *maybe* dependence did not occur. In this case, the necessary value has already been produced locally by this thread and the thread can simply continue executing using this locally produced value.

At the completion of the computation stage, the thread enters its *write-back* stage. If the thread is the nonspeculative head thread, it can write-back all of the values it has produced in the course of its execution to the main memory. It then terminates its execution by sending a synchronization signal to its child thread. Upon receiving this synchronization, the child thread becomes the nonspeculative head thread. The new head thread then enters its write-back stage to transfer the values it temporarily stored in its local memory into the main memory. When a thread terminates and the thread processing unit on which it was executing becomes available, a new thread can be initiated on that thread processing unit as the child of the last thread in the sequence.

The above sequence of operations is what occurs when every thread that was speculatively initiated eventually becomes the nonspeculative head thread. However, one of the powerful features of this threading model is that not all speculative threads need to be allowed to complete their execution. Any thread, once it becomes the head thread, can send an `abort` signal to all of its successor threads. This abort operation causes all of these threads to terminate their execution without writing their speculatively-produced values back to the main memory. The single remaining head thread then continues executing the sequential program code that follows the parallel loop.

For instance, a `while` loop would be parallelized in this model by speculatively initiating an iteration on each of the available thread processing units. Eventually, the current head thread will determine that the loop termination condition has been satisfied. It will then send the `abort` signal to all of its children. Even though these successor threads executed operations that were beyond the limits of the loop's execution, they will have no impact on the program's global state since they stored their speculatively-produced values into their private local memory buffers. Figure 3 shows how the example loop from Figure 1 would be rewritten for this speculative multithreading execution model [12].

```
/* 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 */
```

Fig. 3. The equivalent superthreaded code for the example loop shown in Figure 1.

IV. The Superthreaded Processor

A. Architecture

We have developed the *superthreaded processor architecture* [12], [13], [14] to support the execution of the speculative multithreading model described in the previous section. As shown in Figure 4, each thread processing unit in the overall architecture is composed of a superscalar processor core with some additional components to support the speculative execution of threads. Our target implementation is to construct four to eight thread processing units on a single chip, with the potential to expand to sixteen thread processing units as VLSI technology continues to improve. Each thread processing unit has its own private register file to store the local values it uses in its computations. Each unit also has its own program counter to allow both independent and speculative instruction streams to be executed. There is a single instruction cache shared by all of the units.

The *dependence buffer* in each thread unit provides temporary storage for all values produced by a thread during speculative execution. When a load instruction is executed that the compiler has marked as a *maybe* dependence, this dependence buffer performs an associative search for the potentially matching target store address that may have been forwarded by an upstream thread unit. Thus, this buffer provides the mechanism needed to perform the run-time checking of these *maybe* data dependences. When a thread enters the write-back stage (see Figure 2), the values that have been produced by this thread, but temporarily stored in the dependence buffer, are written back to the globally-shared memory through the common data cache.

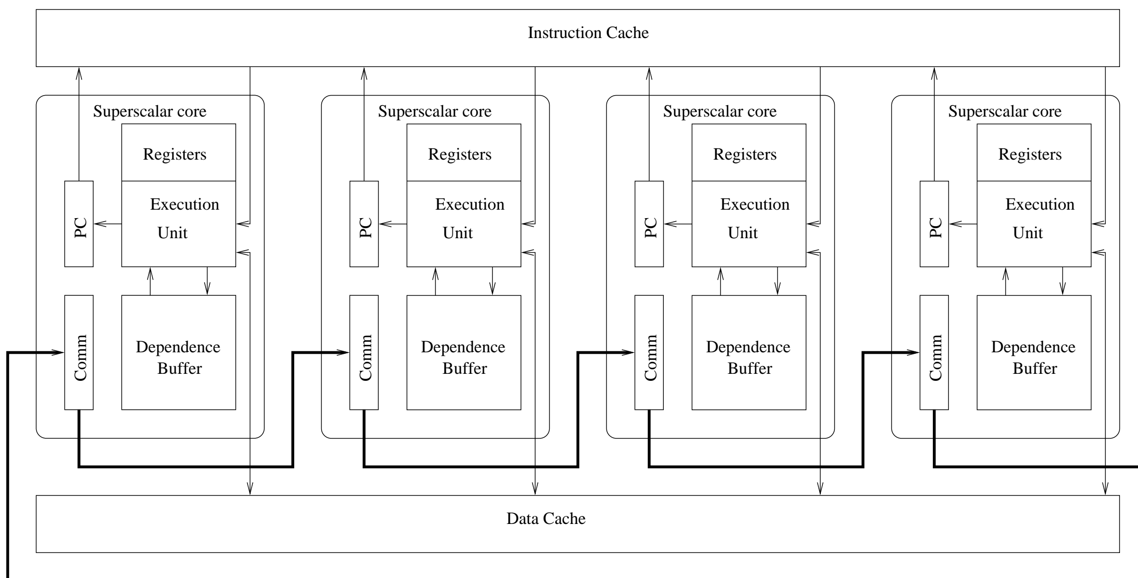

Fig. 4. The basic organization of the superthreaded processor architecture with four thread processing units.

In the thread pipelining model, communication between threads always proceeds from the head thread downstream to its successor threads. As a result, the superthreaded architecture requires only a simple unidirectional ring to support communication between thread units. Notice that the output of the communication unit from the last thread processing unit wraps around to the input of the communication unit in the first thread unit. When the head thread of a parallel loop completes its write-back stage, this circular structure allows the next thread unit in the chain to convert to the nonspeculative execution mode and thereby become the new head thread. The unit that just completed executing the head thread now becomes the last (i.e., the furthest downstream) thread unit in the ring. It then can begin executing the next speculative thread that was previously initiated by its upstream thread unit.

B. Compiler Support

While the architectural features of the superthread processor can simplify the compiler's analysis of pointer aliasing and data dependences, the compiler still must be able to partition threads into the appropriate stages. It also must reorder instructions to maximize the amount of overlap among executing threads. To support the superthreaded processor, we are developing a compiler infrastructure that can exploit speculative thread-level parallelism for both C and Fortran programs.

The initial version of this compiler uses a modified version of the SUIF compiler [15] as the front-end and an enhanced version of the GCC compiler [16] as the back-end. This approach allows us to combine the functionality of an existing parallelizing compiler in the front-end with a sophisticated optimizing compiler in the back-end. This separation is useful since the two compilers have very different requirements for data structures and emphasize different analysis and optimization techniques. However, instead of treating them as two independent compilers, we have developed a scheme that exports high-level information (HLI) from the front-end to the back-end to allow them to work together as a single entity [12], [17]. This HLI contains high-level program information about data dependences, pointer aliases, array dataflow information, and parallel loops. This information is used in the enhanced GCC compiler for aggressive back-end optimizations.

Although the SUIF compiler is very strong in parallel optimizations for Fortran programs, it is somewhat weak in supporting pointer analysis for C programs. Consequently, we are developing a new compiler as part of this project that includes new techniques for pointer alias analysis, data dependence analysis, integrated interprocedural dataflow analysis, and sophisticated loop parallelization techniques [18]. These new analysis and optimization techniques take into consideration the hardware support for speculation and runtime data dependence checking in the superthreaded architecture.

C. Performance Evaluation

We have developed a simulator for the superthreaded processor [19] that we have used to characterize its performance potential [14]. This simulator is based on the SimpleScalar simulator [20] that is commonly used among the computer architecture research community. Our simulation methodology builds a framework around the SimpleScalar simulator to include the components that are unique to the superthread architecture, such as the dependence buffer and the interprocessor communication ring. The components included in this additional framework effectively simulate the *continuation*, *target store address generation*, and *write-back* stages of the speculative multithreading execution model described in Section III. The SimpleScalar simulator then is used to execute the *computation* stage. The superthreaded `fork` instruction that is

executed at the end of the *continuation* stage is simulated by replacing it with an operating system-level fork operation. This operating system fork effectively creates a new copy of the SimpleScalar simulator to use as the simulated thread processing unit for the newly forked thread.

Table I shows the characteristics of the benchmark programs used to evaluate the performance of the superthreaded architecture with this simulator. The programs *wc* and *cmp* are the GNU utility functions *wordcount* and *compare*. The programs *compress*, *ijpeg*, and *m88ksim* are integer programs from the SPEC95 benchmark suite. These five programs are used to evaluate the effectiveness of the superthreaded architecture in parallelizing applications that have traditionally been difficult to parallelize using conventional techniques. The programs *alvinn*, *hydro2d*, and *ear* are floating-point programs taken from the SPEC92 benchmarks. These three programs were chosen to evaluate this architecture's effectiveness on numerical types of application programs. The third column in the table shows the total number of instructions executed by the application on a baseline single-threaded superscalar processor that is capable of issuing two instructions per cycle. The last column shows the performance achieved by this baseline processor measured as the average number of instructions completed per cycle (IPC).

TABLE I
The benchmark programs used to evaluate the superthreaded architecture.

Program	Input Set	Instr count	Baseline IPC
wc	expr.c from GCC	3.1M	0.88
cmp	expr.c from GCC	2.7M	0.87
compress	train	50.6M	0.92
ijpeg	test	1046.1M	0.99
m88ksim	test	972.1M	0.97
alvinn	ref (20 iters)	613.6M	1.07
hydro2d	test	33.9M	0.96
ear	short	812.7M	1.13

To determine the effectiveness of this multithreading model, we evaluated several different configurations of the superthread processor while holding the overall system capability constant. Specifically, we changed the number of thread units from 1 to 16 while simultaneously changing the instruction issue capability within each thread unit from 32 to 2. Thus, the total number of instructions that can be issued and completed in each cycle for every configuration is constant at 32. The difference among the configurations is the number of independent thread units available. The other system parameters are also adjusted proportionally so that there is always a total of 16 integer and 16 floating-point ALUs. The above benchmarks are hand-compiled to approximate the compilation results expected from the real compiler when it is completed.

Figure 5 shows the speedups obtained by the superthreaded processor compared to the two-issue baseline superscalar processor when executing the benchmark programs. The left-hand side of the x-axis in these figures corresponds to a single thread unit capable of issuing 32 instructions per cycle. That is, this configuration represents a single, very wide-issue superscalar processor. It is used to compare the improvement made possible by the superthreading approach. However, this very wide superscalar configuration most likely would be very difficult to implement in practice. The number of thread units increases for points further to the right in this figure, while the number of instructions that can be issued by each thread unit each cycle decreases proportionally. Finally, the right-most point in each figure corresponds to 16 independent thread units, each of which can issue 2 instructions per cycle. Thus, the total number of instructions that can be issued in each cycle is constant at 32.

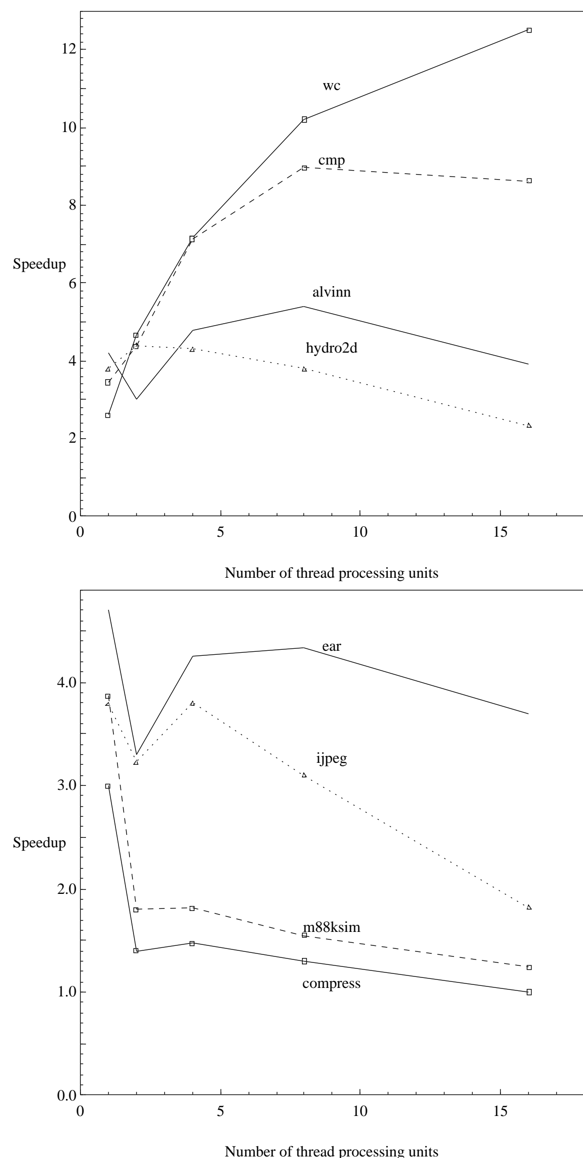

Fig. 5. Simulated speedups for the superthreaded processor relative to the baseline two-instruction-issue superscalar processor.

We observe in these figures that the supertreaded architecture approach improves the performance of *wc*, *cmp*, *alvinn*, and *hydro2d* compared to the single-threaded wide issue superscalar processor. These four programs have relatively high degrees of thread-level parallelism that cannot be easily exploited by the single-threaded superscalar processor. Both *compress* and *m88ksim* show relatively poor speedups compared to the wide issue superscalar processor, however. When executed with multiple thread units, *compress* frequently attempts to execute its threads speculatively. However, it contains several loop-carried *maybe* data dependences that turn out to conflict at run-time. These conflicts require the misspeculated threads to be aborted. This frequent forking and subsequent aborting adds substantial overhead to the execution of this program without providing any compensating speedup due the parallel execution of the threads. While *m88ksim* does not abort its speculative threads as frequently as *compress*, its threads often do not contain enough work to compensate for the thread creation overhead.

The performance of the four thread unit configuration for *ijpeg* is approximately the same as the performance of the single-threaded configuration. As the number of thread units increases, however, the overhead per thread remains roughly constant, while a constant amount of work gets spread over a larger number of thread units. As a result, the impact of the parallelization overhead (i.e., the forking and synchronization operations) reduces the overall performance gain for this program.

The performance obtained for *ear* on both the four and eight thread unit configurations is slightly below that obtained on the single superscalar configuration. However, given the difficulty of building such a wide superscalar configuration, the superthreaded organization would probably be the best choice for this application. Taking into consideration both their relative performance and their expected implementation complexities, we conclude that the best overall configuration is likely to be a superthreaded processor with four to eight thread units, each of which is capable of issuing two to four instructions per cycle.

D. Coarse-Grained Speculative Multithreading

As mentioned in Section IV-C, the simulator for the superthreaded processor is built on top of the SimpleScalar simulator. We use operating system `fork` operations to begin executing an essentially new copy of the SimpleScalar simulator whenever a simulated superthreaded processor `fork` instruction is encountered. This approach allowed us to quickly develop an execution-driven simulator to evaluate this new architecture [14]. We subsequently observed that we could replace the SimpleScalar simulator with native machine code to execute the *computation* stage. While this change would eliminate the ability to obtain clock-cycle accurate performance estimates of the superthreaded processor, it would instead allow us to more quickly execute an application that had been compiled to use the superthreaded instruction set.

To exploit this speculative execution model on an existing computer system, we developed special C language library functions [21] that the compiler or an application programmer can insert into the source code to simulate the behavior of the superthreaded parallelization instructions. This then allows traditionally sequential program constructs, such as `while` loops, to be executed concurrently on standard, off-the-shelf multiprocessor systems. We have evaluated the performance of application programs that use these library functions on an eight-processor Silicon Graphics shared-memory multiprocessor system. We found that this approach obtained better speedups on these applications than the most competitive alternative approaches [22], [23], [24]. We also have developed corresponding library functions for programs written in the Java programming language [25]. These libraries now allow the speculative execution of programs that are parallelized using the above superthreaded execution model on any shared-memory multiprocessor system that supports either C or Java.

V. Related Work

A variety of multithreaded architectures have been proposed both to tolerate long memory delays and to increase the total number of instructions that can be issued in each cycle. The HEP [26], Horizon [27] and Tera [28] machines, for instance, maintain hundreds of thread contexts in each processor to allow switching between threads every cycle. With no data caches, this approach allows the processors to tolerate long memory delays. Instead of switching contexts each cycle, a processor in the Alewife machine [29] executes instructions from a single thread until the thread issues an instruction that causes a long-latency memory access. Alewife also tries to avoid memory delays by using a data cache in addition to context switching.

To increase parallelism, the XIMD [30], Elementary Multithreading [31], M-machine [32], Simultaneous Multithreading [33], Multiscalar [34], and SPSM [35] architectures execute instructions from multiple threads simultaneously. The XIMD, Elementary Multithreading, M-machine, and Multiscalar architectures support synchronization and communication between threads to allow the execution of loops with dependences between iterations. The SPSM and the Multiscalar also allow speculative execution of threads. The Multiscalar further supports speculation on data dependences using the globally-shared *Address Resolution Buffer* (ARB). This hardware structure automatically checks for read-write ordering violations among concurrently executing speculative threads. If the ARB detects a violation, it will force all of the speculative threads to be aborted. These threads then must be restarted to use the correct values.

Another dimension along which multithreaded architectures can be compared is the level of compiler support they require. The ATLAS [36], Multiscalar [34], and Raw [37] architectures, for example, all expect the compiler to perform different degrees of dependence analysis and program optimization when compared to the superthreaded architecture. As suggested in Figure 6, the ATLAS architec-

ture makes all thread partitioning decisions and resolves all ambiguous dependences dynamically at run-time. At the other extreme, the Raw architecture requires extensive compiler support to execute any application program. This architecture replicates very simple processors, called *tiles*, onto a single chip with all of the low-level details exposed to the compiler. The compiler then programs the tiles to match the needs of a specific application. For example, if the compiler determines that an application would benefit from a certain systolic array organization, it programs the tiles into this configuration specifically for the execution of that single application.

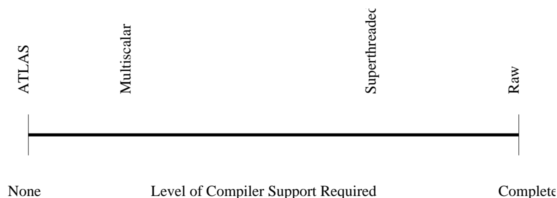

Fig. 6. Comparing the level of compiler support needed by different multithreaded architectures.

The multiscalar and superthreaded architectures choose a more balanced approach between using only run-time analysis and requiring complete compile-time analysis. While the Multiscalar uses extensive hardware support to dynamically analyze and correct read-write violations, it does require the compiler to partition the application program into appropriate threads [38]. Our approach with the superthreaded processor has been to simplify the hardware as much as possible compared to the Multiscalar or the ATLAS architectures. The superthreaded processor then relies on the compiler to partition the application program into threads. Additionally, the compiler must identify dependences that may exist at run-time (i.e., *maybe* dependences) and must pass this information to the hardware using the target store operations. We feel that this approach greatly simplifies the implementation of this type of multithreaded architecture compared to the hardware-only approaches. It also allows the compiler to make more aggressive optimization assumptions than would otherwise be possible without the hardware-supported control speculation among threads and the run-time checking of data dependences.

VI. Conclusion

A. Summary

Computer designers have three primary handles with which to improve performance: 1) using faster VLSI technology and deep pipelining to improve the clock rate; 2) reducing the total number of instructions executed by adjusting the compiler-architecture interface; and 3) increasing the number of instructions executed in each cycle. This paper has summarized our recent work in developing the superthreaded processor architecture as an approach for increasing the number of instructions executed per cycle instead of simply increasing the issue width of a conventional superscalar processor design. One of the key features of this approach is to provide additional hardware support that allows the compiler to make the most aggressive assumptions possible about *maybe* dependences, which are dependences that cannot be determined completely at compile-time using only static information. The superthreaded processor supports speculation on control dependences by allowing threads to be initiated before it is known whether they actually should be executed. A special-purpose dependence buffer is also provided to allow the run-time checking of *maybe* dependences.

We have evaluated the performance of this new architecture using a detailed cycle accurate execution-driven simulator. We find that, for many application programs that are difficult to parallelize using conventional approaches, the superthreaded architecture can obtain substantially better performance than a single-threaded wide-issue superscalar processor. For some application programs that have only relatively fine-grained parallelism, though, the wide-issue superscalar processor outperforms the superthreaded processor. However, taking into account the implementation complexity of a very wide-issue superscalar processor, we conclude that a superthreaded processor with four to eight thread units, each of which is capable of issuing from two to four instructions per cycle, is likely to be the best overall configuration. We also briefly described program libraries for C and Java that extend the superthreaded execution model to conventional off-the-shelf shared-memory multiprocessors.

B. Future Directions

Our preliminary performance analyses have made clear the importance of sophisticated compiler optimizations targeted specifically to this type of speculative multithreaded architecture. Consequently, we currently have an extensive effort in progress to develop compile-time optimizations to support and enhance speculative execution [18]. We also are concerned about the implementation cost of this type of multithreaded architecture, as measured by the chip area necessary to implement the thread processing units and the necessary shared resources. To study the feasibility of its implementation, we are currently developing a detailed gate-level model of a superthreaded processor with four thread units using the Verilog hardware description language. This model will allow us to determine actual gate counts for the various subsystems, and will allow us to determine expected signal delays for specific VLSI technology libraries. Knowing these delays then will allow us to determine the fastest clock cycle that could be used in this processor.

We conclude that this type of speculative multithreaded architecture has the potential to substantially improve the execution time performance of application programs that have been hard to parallelize using conventional approaches. However, considerable work remains to be done before we can determine the feasibility of this approach for general-purpose microprocessor designs of the future.

Acknowledgements

This paper summarizes the work of many of my students and colleagues. In particular, Jenn-Yuan Tsai and Pen-Chung Yew first proposed the basic ideas behind the superthreaded architecture [13]; Zhiyuan Li made numerous important contributions to the compiler; and the following current and former students did much of the actual implementation and evaluation work: C. Amlo, S. Cho, J. Huang, Z. Jiang, I. Kazi, S. Schwinn, Y. Song, X. Wang, Q. Zhao, and B. Zheng. This work has been supported by National Science Foundation grants MIP-9610379 and EIA-9971666; by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order No. D 346; and by a gift from the Intel Corporation.

References

[1] Per Stenstrom, Erik Hagersten, David J. Lilja, Margaret Martonosi, , and Madan Venogupal, “Trends in shared-memory multiprocessing,” in *IEEE Computer*, December 1997, vol. 30, pp. 44–50.

[2] Luiz Andre Barroso, Kourosh Gharachorloo, and Edouard Bugnion, “Memory system characterization of commercial workloads,” in *International Symposium on Computer Architecture*, 1998.

[3] Z. Cvetanovic and D. Bhandarkar, “Characterization of Alpha AXP performance using TP and SPEC workloads,” in *International Symposium on Computer Architecture*, 1994, pp. 60–70.

[4] M. T. Franklin, W. P. Alexander, R. Jauhari, A. M. G. Maynard, and B. R. Olszewski, “Commercial workload performance in the IBM POWER2 RISC System/6000 processor,” in *IBM Journal of Research and Development*, September 1994, vol. 38, pp. 555–561.

[5] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski, “Contrasting characteristics and cache performance of technical and multi-user commercial workloads,” in *ASPLOS*, 1994, pp. 145–156.

[6] P. Trancoso, J. Larriba-Pey, Z. Zhang, and J. Torrellas, “Memory performance of DSS commercial workloads in shared-memory multiprocessors,” in *International Symposium on High-Performance Computer Architecture*, 1997, pp. 250–260.

[7] Stephanie Coleman and Kathryn S. McKinley, “Tile size selection using cache organization and data layout,” in *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 18–21, 1995, pp. 279–290.

[8] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf, “The cache performance and optimizations of blocked algorithms,” in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 8–11, 1991, pp. 63–74.

[9] Steven VanderWiel and David J. Lilja, “When caches are not enough: Data prefetching techniques,” in *IEEE Computer*, July 1997, vol. 30, pp. 23–30.

[10] David J. Lilja, *Measuring Computer Performance: A Practitioner's Guide*, Cambridge University Press, Cambridge, United Kingdom, 2000.

[11] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, Inc., San Francisco, 1996.

[12] Jenn-Yuan Tsai, Zhenzhen Jiang, Zhiyuan Li, David J. Lilja, Xin Wang, Pen-Chung Yew, Bixia Zheng, and Stephen J. Schwinn, “Integrating parallelizing compilation technology and processor architecture for cost-effective concurrent multithreading,” in *Journal of Information Science and Engineering*, March 1998, vol. 14.

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

[14] Jenn-Yuan Tsai, Jian Huang, Christoffer Amlo, David J. Lilja, and Pen-Chung Yew, “The superthreaded processor architecture,” in *IEEE Transactions on Computers, Special Issue on Multithreaded Architectures and Systems*, September 1999, pp. 881–902.

[15] S. Tjiang, M. Wolf, M. Lam, K. Pieper, and J. Hennessy, “Integrating scalar optimization and parallelization,” in *Proceedings of the Workshop on Languages and Compilers for Parallel Computers*, 1991, pp. 137–151.

[16] R. M. Stallman, *Using and Porting GNU CC (version 2.7)*, Free Software Foundation, Cambridge, Massachusetts, 1995.

[17] Sangyeun Cho, Jenn-Yuan Tsai, Yonghong Song, Bixia Zheng, Stephen Schwinn, Xin Wang, Qing Zhao, Zhiyuan Li, David J. Lilja, and Pen-Chung Yew, “High-level information: An approach for integrating front-end and back-end compilers,” in *International Conference on Parallel Processing*, August 1998, pp. 246–355.

[18] B. Zheng, J.-Y. Tsai, B. Y. Zang, T. Chen, B. Huang, J. H. Li, Y. H. Ding, J. Liang, Y. Zhen, P.-C. Yew, and C.Q. Zhu, “Designing the Agassiz compiler for concurrent multithreaded architectures,” in *Workshop on Languages and Compilers for Parallel Computing*, August 1999.

[19] Jian Huang and David J. Lilja, “An efficient strategy for developing a simulator for a novel concurrent multithreaded processor architecture,” in *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 1998.

[20] Doug Burger and Todd M. Austin, “The SimpleSclar tool set, version 2.0,” in *University of Wisconsin-Madison Computer Science Department Technical Report no. 1342*, June 1997.

[21] Iffat H. Kazi and David J. Lilja, “Coarse-grained speculative execution in shared-memory multiprocessors,” in *International Conference on Supercomputing*, July 1998, pp. 93–100.

[22] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua, “Run-time methods for parallelizing partially parallel loops,” in *International Conference on Supercomputing*, July 3–7, 1995, pp. 137–146.

[23] Lawrence Rauchwerger and David Padua, “The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization,” in *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 18–21, 1995, pp. 218–232.

[24] L. Rauchwerger and D. Padua, “The privatizing doall test: A run-time technique for doall loop identification and array privatization,” in *SIGPLAN Conference on Supercomputing*, July 1994, pp. 33–43.

[25] Iffat H. Kazi and David J. Lilja, “JavaSpMT: A speculative thread pipelining parallelization model for Java programs,” in *International Parallel and Distributed Processing Symposium*, May, 2000, pp. 559–564.

[26] Harry F. Jordan, “Performance measurements on HEP — a pipelined MIMD computer,” in *Proceedings of the 10th Annual International Symposium on Computer Architecture*, June 13–17, 1983, pp. 207–212.

[27] J. T. Kuehn and B. J. Smith, “The Horizon supercomputing system: Architecture and software,” in *Proceedings of Supercomputing 1988*, November 1988.

[28] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith, “The Tera computer system,” in *1990 International Conference on Supercomputing*, June 11–15, 1990, pp. 1–6.

[29] A. Agarwal, D. Chaiken, G. D'Souza, K Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. A. Yeung, “The MIT Alewife machine: A large-scale distributed-memory multiprocessor,” in *Proceedings of Workshop Multithreaded Computers, Supercomputing 91*, November 1991.

[30] 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*, April 8–11, 1991, pp. 2–14.

[31] 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*, May 19–21, 1992, pp. 136–145.

[32] Marco Fillo, Stephen W. Keckler, William J. Dally, 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, November 29–December 1, 1995, pp. 146–156.

[33] 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*, June 22–24, 1995, pp. 392–403.

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

[35] 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*, June 27–29, 1995, pp. 109–121.

[36] Lucian Codrescu and D. Scott Wills, "Architecture of the Atlas chip-multiprocessor: Dynamically parallelizing irregular applications," in *International Conference on Computer Design*, October, 1999.

[37] Elliot Waingold and et al, "Baring it all to software: Raw machines," in *IEEE Computer*, September 1997, pp. 86–93.

[38] T. N. Vijaykumar and Gurindar S. Sohi, "Task selection for the multiscalar architecture," in *Journal of Parallel and Distributed Computing, Special Issue on Compilation and Architectural Support for Parallel Applications*, August, 1999.