Multiscalar Processors

Gurindar S. Sohi
sohi@cs.wisc.edu

Scott E. Breach
breach@cs.wisc.edu

T.N. Vijaykumar
vijay@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

Abstract

Multiscalar processors use a new, aggressive implementation paradigm for extracting large quantities of instruction level parallelism from ordinary high level language programs. A single program is divided into a collection of tasks by a combination of software and hardware. The tasks are distributed to a number of parallel processing units which reside within a processor complex. Each of these units fetches and executes instructions belonging to its assigned task. The appearance of a single logical register file is maintained with a copy in each parallel processing unit. Register results are dynamically routed among the many parallel processing units with the help of compiler-generated masks. Memory accesses may occur speculatively without knowledge of preceding loads or stores. Addresses are disambiguated dynamically, many in parallel, and processing waits only for true data dependences.

This paper presents the philosophy of the multiscalar paradigm, the structure of multiscalar programs, and the hardware architecture of a multiscalar processor. The paper also discusses performance issues in the multiscalar model, and compares the multiscalar paradigm with other paradigms. Experimental results evaluating the performance of a sample of multiscalar organizations are also presented.

1. Introduction

The basic paradigm of sequencing through a program, *i.e.*, the fetch-execute cycle using a program counter, has been with us for about 50 years. A consequence of this sequencing paradigm is that programs are written with the tacit assumption that instructions will be executed in the same order as they appear in the program. To achieve high performance, however, modern processors attempt to execute multiple instructions simultaneously, and in some cases in a different order than the original program sequence. This reordering may be done in the compiler, in the hardware at execution time, or both. Superscalar and VLIW processors belong to this class of architectures that exploit instruction level parallelism (ILP).

ILP processors and compilers typically convert the total ordering of instructions as they appear in the original program into a *partial ordering* determined by dependences on data and control. Control dependences (which appear as conditional branches) present a major obstacle to highly parallel execution because these dependences must be resolved before all subsequent instructions are known to be valid.

Focusing on control dependences, one can represent a static program as a *control flow graph (CFG)*, where basic blocks are nodes, and arcs represent flow of control from one basic block to another. Dynamic program execution can be viewed as walking through the program CFG, generating a dynamic sequence of basic blocks which have to be executed for a particular run of the program.

To achieve high performance, an ILP processor must attempt to walk through the CFG with a high level of parallelism. Branch prediction with speculative execution is one commonly-used technique for raising the level of parallelism that can be achieved during the walk. The primary constraint on any parallel walk, however, is that it must preserve the sequential semantics assumed in the program.

In the multiscalar model of execution, the CFG is partitioned into portions called tasks. A multiscalar processor walks through the CFG speculatively, taking task-sized steps, without pausing to inspect any of the instructions within a task. A task is assigned to one of a collection of processing units for execution by passing the initial program counter of the task to the processing unit. Multiple tasks then execute in parallel on the processing units, resulting in an aggregate execution rate of multiple instructions per cycle.

At this level, the concept sounds simple, however, the key to making it work is the proper resolution of inter-task data dependences. In particular, data that is passed between instructions via registers and memory must be routed correctly by the hardware. Furthermore, it is in this area of inter-task data communication that the multiscalar approach differs significantly from more traditional multiprocessing methods.

This paper describes the multiscalar approach to exploiting fine-grain parallelism (or instruction-level parallelism or ILP). Section 2 provides an overview of the multiscalar paradigm. A breakdown of the distribution of the available processing unit cycles in multiscalar execution follows in Section 3. In Section 4, we compare multiscalar with other ILP paradigms. A performance evaluation of potential configurations of a multiscalar processor is given in Section

5. In Section 6, we summarize this work and offer concluding remarks.

2. An Overview of the Multiscalar Paradigm

2.1. Philosophy and Basics

The objective of the non-sequential walk of the CFG taken by a multiscalar processor is to establish a large and accurate dynamic window of instructions from which independent instructions can be extracted and scheduled for parallel execution. (An instruction window, in ILP parlance, is an assemblage of instructions under consideration for execution.) To perform this function, a multiscalar processor walks through the CFG in large steps, not instruction by instruction (as is the case in a sequential processor), nor basic block by basic block, but rather task by task.

A task is a portion of the CFG whose execution corresponds to a contiguous region of the dynamic instruction sequence (*e.g.*, part of a basic block, a basic block, multiple basic blocks, a single loop iteration, an entire loop, a function call, etc.). A program is statically partitioned into tasks which are demarcated by annotations of the CFG (more on this in Section 2.2). For each step of its walk, a multiscalar processor assigns a task to a processing unit for execution, without concern for the actual contents of the task, and continues its walk from this point to the next point in the CFG.

A possible microarchitecture for a multiscalar processor is shown in Figure 1. In most general terms, consider a multiscalar processor to be a collection of processing units with a sequencer which assigns tasks to the processing units. Once a task is assigned to a processing unit, the unit fetches and executes the instructions of the task until it is complete. Multiple processing units, each with its own internal instruction sequencing mechanism, support the execution of multiple tasks, and thereby multiple instructions, in any given time step. The instructions contained within the dynamic instruction window are bounded by the first instruction in the earliest executing task and the last instruction in the latest executing task. Given that each task may contain loops and function calls, this observation implies that the effective size of the instruction window may be extremely large. A key point is that not all the instructions within this wide range are simultaneously being considered for execution, only a limited set within each of the processing units.

Consider the CFG in Figure 2 of a program fragment with five basic blocks, *A*, *B*, *C*, *D*, and *E*. Suppose the dynamic sequence of basic blocks executed is $A_1^1\ B_1^1\ C_1^1\ B_2^1\ B_3^1\ C_2^1\ D_1^1\ \ A_1^2\ B_1^2\ B_2^2\ C_1^2\ D_1^2\ \ A_1^3\ B_1^3\ C_1^3\ B_2^3\ C_2^3\ D_1^3\ \ E$. In this sequence, the superscripts and subscripts identify the incarnation of the basic block in relation to the outer and inner loops, respectively. In a sequential processor, the dynamic instructions corresponding to this sequence of basic blocks are generated as program control navigates through the CFG, executing one instruction at a time. To ensure a correct execution on an ILP processor, it must *appear* that the instructions among all basic blocks execute in precisely this same sequential order, regardless of what actually transpires.

Consider an iteration of the outer loop from the CFG in Figure 2 as a task. That is, let static basic blocks *A*, *B*, *C*, and *D* (as well as the control flow through them) comprise a task. We may assign a task corresponding to the first iteration of the outer loop to a processing unit, followed by the second iteration to the next processing unit, and so on.

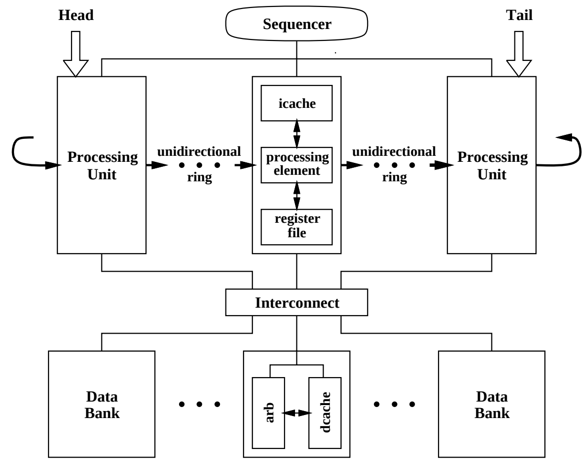

Figure 1: A Possible Microarchitecture of a Multiscalar Processor.

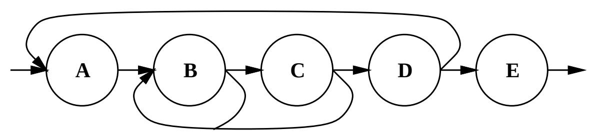

Figure 2: An Example Control Flow Graph.

The processing unit that is assigned the first iteration sequences through its task to execute the dynamic instructions of basic blocks $A_1^1\ B_1^1\ C_1^1\ B_2^1\ B_3^1\ C_2^1\ D_1^1$. Likewise, the following processing units execute the dynamic instructions of basic blocks $A_1^2\ B_1^2\ B_2^2\ C_1^2\ D_1^2$ and $A_1^3\ B_1^3\ C_1^3\ B_2^3\ C_2^3\ D_1^3$, as per the second and third iterations respectively. In this example, the potential result of this approach is the execution of three useful instructions in a cycle. For instance, in a given cycle, the processing units might execute instructions from dynamic basic blocks B_2^1, C_1^2, and B_2^3, simultaneously.

It is important to observe that tasks, although separate groups of instructions, are not independent. Because tasks are portions of a sequential instruction stream, the data and control relations among individual instructions must be honored during execution. A key issue in a multiscalar implementation is the communication of data and control information among the parallel processing units. That is, how do we provide the appearance of a sequential walk even though in reality we perform a non-sequential walk (perhaps considered radically non-sequential) through the CFG?

To maintain a sequential appearance we employ a twofold strategy. First, we ensure that each processing unit adheres to sequential execution semantics for the task assigned to it. Second, we enforce a loose sequential order over the collection of processing units, which in turn imposes a sequential order on the tasks. The sequential order on the processing units is maintained by organizing the units into a circular queue. Head and tail pointers indicate the units that are executing the earliest and the latest of the current tasks, respectively. For instance in the example of Figure 2, the processing unit at the head is executing the first iteration, preceding the unit executing the second iteration, preceding the tail unit executing the third iteration.

As instructions in a task execute, values are both consumed and produced. These values are bound to storage locations, namely registers and memory. Because a sequential execution model views storage as a single set of registers and memory locations, multiscalar execution must maintain this view as well. Furthermore, multiscalar execution must ensure that the values consumed and produced by instructions are the same as those in a sequential execution. In the example, values consumed by an instruction in dynamic basic block B_2^2 must be the values resulting from the execution of instructions in A_1^1 B_1^1 C_1^1 B_2^1 B_3^1 C_2^1 D_1^1 A_1^2 B_1^2, as well as preceding instructions in B_2^2. In order to provide this behavior, we must synchronize communication between tasks.

In the case of registers, the control logic synchronizes the production of register values in predecessor tasks with the consumption of these values in successor tasks via reservations on registers. The register values a task may produce can be determined statically and maintained in a *create mask* (more details in Section 2.2). At the time a register value in the create mask is produced, it is forwarded to later tasks, *i.e.*, to processing units which are logical successors of the unit, via a circular unidirectional ring (see Figure 1). The reservations on registers for a successor task are given in the *accum mask*, which is the union of the create masks of currently active predecessor tasks. As values arrive from predecessor units, reservations are cleared in the successor units. If a task uses one of these values, the consuming instruction can proceed only if the value has been received; otherwise it waits for the value to arrive.

In the case of memory, the situation is somewhat different. Unlike register values, it cannot be precisely determined ahead of time which memory values are consumed or produced by a task. If it is known that a task consumes a memory value (via a load instruction) that is produced (via a store instruction) in an earlier task, it is possible to synchronize the consumption and production of this value. That is, the load in the successor task can be made to wait until the store in the predecessor task has completed (similar in concept to the situation for registers, although the exact synchronization mechanism would be different due to the disparity in the sizes of the name-spaces).

In the more common case where such knowledge is not available, either a conservative or an aggressive approach may be undertaken. The conservative approach is to wait until it is certain that the load will read the correct value. This option typically implies holding back loads within a task until all predecessor tasks have completed all stores, with the likely outcome being near-sequential execution. The aggressive approach is to perform loads speculatively, with the expectation that a predecessor task will not store a value into the same location at a later time. A check must be made dynamically to ensure that no predecessor task writes a value into a memory location previously read by a successor task. If this check identifies a load and store that conflict (do not occur in the proper order), the later task must squash its execution and initiate appropriate recovery action. (A multiscalar processor takes the aggressive approach.)

Due to the speculative nature of multiscalar execution, it must be possible to both confirm correct execution as well as recover from incorrect execution. The execution of instructions within tasks may be considered as speculative for two reasons: (i) control speculation, and (ii) data speculation. As tasks execute, the correct path of execution through the program CFG is resolved. If control speculation, *i.e.*, prediction of the next task, is incorrect, the following task(s) must be squashed and the correct task sequence resumed. Likewise, if a task uses an incorrect data value, the offending task must be squashed and the correct data value recovered. In any case, the action of squashing a task results in the squashing of all tasks in execution following the task (otherwise, maintaining sequential semantics becomes complex).

To facilitate maintaining sequential semantics, a multiscalar processor retires tasks from the circular queue of units in the same order as it assigns them. During speculative execution, a task produces values which may or may not be correct. It is only certain the values produced by a task are correct, and may be consumed safely by other tasks, at the time the retirement of a task is imminent. Nevertheless, values are optimistically forwarded for speculative use throughout the execution of a task. Because a task forwards values to other tasks as it produces them (more details in Section 2.2 and Section 2.3), most, if not all, of its values have been forwarded by the time it becomes the head. Thus, retiring the task may simply be a matter of updating the head pointer to free the processing unit so a new task may be assigned.

To illustrate the power of the multiscalar model of execution, consider the example in Figure 3. In this code segment, execution repeatedly takes a symbol from a buffer and runs down a linked list checking for a match of the symbol. If a match is found, a function is called to process the symbol. If no match is found, an entry in the list is allocated for the new symbol. After an initial startup, additions to the list become infrequent, because most symbols match an element already in the list. In a multiscalar execution, a task assigned to a processing unit comprises one complete search of the list with a particular symbol. The processing units perform a search of the linked list in parallel, each with a

```
for (indx = 0; indx < BUFSIZE; indx++) {
    /* get the symbol for which to search */
    symbol = SYMVAL(buffer[indx]);

    /* do a linear search for the symbol in the list */
    for (list = listhd; list; list = LNEXT(list)) {
        /* if symbol already present, process entry */
        if (symbol == LELE(list)) {
            process(list);
            break;
        }
    }

    /* if symbol not found in the list, add to the tail */
    if (!list) {
        addlist(symbol);
    }
}
```

Figure 3: An Example Code Segment.

symbol, resulting in an overall execution of multiple instructions per cycle. The details of the parallel execution of what at first appears to be a serial program are presented throughout the rest of this paper.

2.2. Multiscalar Programs

A multiscalar program must provide the means to support a fast walk (through the CFG) that distributes tasks en masse to processing units. Below, we describe three distinct types of information maintained within a machine-level multiscalar program to facilitate this end: (i) the actual code for the tasks which comprises the work, (ii) the details of the structure of the CFG, and (iii) the communication characteristics of individual tasks.

The specification of the code for each task is routine. A task is specified as a set of instructions, in the same fashion as a program fragment for a sequential machine. Although the instruction set architecture (ISA) in which the code is represented affects the design of each individual processing unit, it has little influence on the rest of the design of a multiscalar processor. Hence, the instruction set used to specify the task is of secondary importance. (The significance of this fact is that an existing ISA may be used without a major overhaul.)

The sequencer of a multiscalar processor requires information about the program control flow structure to facilitate a rapid traversal of the CFG. In particular, it needs to know which tasks are possible successors of any given task in the CFG. The multiscalar sequencer uses this information to predict one of the possible successor tasks and to continue the CFG walk from this point. (Unlike the corresponding case in a sequential execution, control proceeds to a successor task before the current task is complete.) Such information can be determined statically and placed in a *task descriptor*. The task descriptors may be interspersed within the program text (for instance, before the code of the task) or placed in a single location beside the program text (for instance, at the end).

To coordinate execution among different tasks, it is necessary to characterize each task according to the set of values that may be consumed by the task and the set of values that may be produced by the task. In a sequential execution, this information is discovered during the instruction decode process as instructions are fetched and inspected. However, the objective in a multiscalar execution is to assign a task to a processing unit and to proceed to the next task without inspecting the contents of the assigned task.

The procedure to handle register values is straightforward. (Memory values are handled as described in Section 2.3.) A static analysis of the CFG is performed by the compiler to supply the *create mask* that indicates the register values a task may produce[1]. A natural location for the create mask is within the task descriptor. Since a task may contain multiple basic blocks whose execution is governed by (dynamically resolved) control conditions, it is not possible to determine statically which register values *will* be created dynamically. As such, the create mask must be conservative, and thereby includes all register values that *may* be produced.

As a processing unit executes the instructions in a task, register values are produced which must be forwarded to succeeding tasks. Because the unit cannot determine *a priori* which instructions comprise its assigned task (the instructions may not even have been fetched), it cannot know which instruction performs the update to a register that must be forwarded to other tasks. In accordance with sequential semantics, only the last update of a register in the task should be forwarded to other tasks. The option exists to wait until all instructions in a task have been executed (*i.e.*, no further updates of registers are possible). However, this strategy is not expedient since it often implies that other tasks must wait, possibly a considerable period of time, for a value that is already available.

The compiler, on the other hand, has knowledge of the last instruction in a task to update a particular register. It can mark this instruction as a special (operate-and-forward) instruction that, in addition to carrying out the specified operation, forwards the result to following processing units. Furthermore, as a unit executes the instructions of its task, it can identify those registers for which values are not going to be produced (although statically it appeared a value might be produced). By virtue of the fact that later tasks must wait for any register that an earlier task indicates it might produce (regardless of whether a value is actually produced), it is necessary to *release* such registers in order to continue execution. When a register is released, the value is forwarded to later units.

For the same reasons a processing unit cannot determine which dynamic instructions comprise its assigned task, it likewise cannot determine *a priori* on which instruction a task will complete, *i.e.*, at what point control flows out of the task. At the time the CFG is partitioned by the compiler, the boundaries of a task and the control edges leaving the task are known. An instruction at one of these exiting edges may be marked with special stopping conditions so that at the time such an instruction is encountered by the processing unit the appropriate conditions can be evaluated. If the stopping conditions associated with the instruction are satisfied, the task is complete.

The specification of forwarding and stopping information is best viewed as the addition of a few tag bits (*forward* and *stop* bits, respectively) to each instruction in a task. Nevertheless, it may be necessary to implement these tag bits differently if the basic ISA is not to be changed. One possible implementation is to provide a table of tag bits to be associated with each static instruction. As the hardware fetches the instructions from the program text and the corresponding tag bits from the table, it concatenates the pair to produce a new instruction. The new instructions can be maintained in the instruction cache, so that the overhead of accessing two memory locations (one for the instructions and one for the bits) is incurred only in the case of a cache miss. The release of a register may be indicated by adding a special *release* instruction to the base ISA or by overloading an existing instruction of the base ISA.

---

[1] It is not strictly required to specify which values a task may consume. As a task executes and consumes values, it waits for a particular value only if the value has not yet been produced (by an active predecessor task). Otherwise, it finds the value within local storage [1]. The value present within local storage is the product of an earlier task that has forwarded a value around the ring.

A pictorial representation of the information assembled within a task of a multiscalar program is given in Figure 4. This depiction centers around the assembly language for the example of Figure 3. In addition to the assembly language, the figure contains a task descriptor, a set of forward bits, and a set of stop bits. Recall that the task under consideration consists of one iteration of the outer loop. The task executes the iterations of the inner loop to search for a match of a symbol in a linked list. If a match is found, a function is called to process the symbol. If no match is found, an entry in the list is allocated for the symbol. Thus, the task has two possible successor tasks, both of which are the targets of a branch instruction. The successor tasks are either the next iteration of the outer loop (Targ1 = OUTER), or an exit from the outer loop (Targ2 = OUTERFALLOUT). The task completes when the end of the outer loop is reached. Consequently, the last instruction in the outer loop is tagged with a set of bits which indicate a ''Stop Always'' condition.

The task creates values that are bound to the registers: $4, $8, $17, $20, $23. The last instruction to write into registers $4, $20, and $23 has a forward bit set. Since $8 and $17 are updated repeatedly in the inner loop, and only the last update needs to be forwarded, the registers are released at the exit of the inner loop. Along the same lines, $4 is released if the inner loop is skipped, since the instruction that last updates and forwards $4 (in the inner loop) is not executed in this case. It should be noted that a value bound to a register is only sent once per task. Hence, all subsequent forwards and releases of a value already forwarded or released are ignored. To illustrate, the release of $4 is encountered (and ignored) if the instruction that last updates and forwards $4 (in the inner loop) is executed. (An alternative to this approach, which may have the undesirable effect of creating complex intra-task control structures, is to incorporate additional basic blocks to eliminate such a scenario.)

So far in our discussion we have assumed that all values which are created by a task are communicated to other tasks. To maintain program semantics, however, we do not need to communicate *all* values created by a task. Rather, only values that are potentially live outside a task, *i.e.*, are not *dead* at the end of a task, need to be communicated. Going back to the example of Figure 3, we can see that the only register value that is live outside the task is the induction variable, $20; only $20 must appear in the create mask. No other register value needs to be forwarded, and no release instructions need be present. Furthermore, any stores made to the stack frame inside the *process* function need not be communicated to later tasks. Since the live ranges of registers are already known to a compiler, incorporating dead register analysis is fairly straightforward. At the time of writing of this paper, we are still investigating the subject of dead memory value analysis.

A multiscalar program may be generated from an existing binary by augmenting the binary with task descriptors and tag bits. This multiscalar information may be located within or perhaps to the side of the program text. The job of migrating a multiscalar program from one generation to another generation of hardware might be as simple as taking an old binary, determining the CFG (a routine task), deciding upon a task structure, and producing a new binary. The old multiscalar information is removed and replaced by new multiscalar information to form an updated version of the binary. The core of the binary, however, the fundamental instructions which describe the work of each task remain virtually the same. Only multiscalar specific instructions and any adjustments to relative addresses need be accommodated. This approach bodes well for a smooth software growth path from one hardware generation to the next, especially if recompilation from the source code is not practical.

```
Targ Spec     Branch, Branch
Targ1         OUTER
Targ2         OUTERFALLOUT
Create mask   $4,$8,$17,$20,$23

                                         Forward Bits   Stop Bits
OUTER:
        addu     $20, $20, 16                 F
        ld       $23, SYMVAL−16($20)          F
        move     $17, $21
        beq      $17, $0, SKIPINNER
INNER:
        ld       $8, LELE($17)
        bne      $8, $23, SKIPCALL
        move     $4, $17
        jal      process
        jump     INNERFALLOUT
SKIPCALL:
        ld       $17, NEXTLIST($17)
        bne      $17, $0, INNER
INNERFALLOUT:
        release  $8, $17
        bne      $17, $0, SKIPINNER
        move     $4, $23                      F
        jal      addlist
SKIPINNER:
        release  $4
        bne      $20, $16, OUTER                         Stop Always
OUTERFALLOUT:
```

Figure 4: An Example of a Multiscalar Program.

2.3. Multiscalar Hardware

The function of the multiscalar hardware is to walk through the CFG, assign tasks to the processing units, and execute these tasks with the appearance of sequential execution. The job of determining the order of the tasks is the responsibility of the sequencer. Given the address of a task descriptor, the sequencer fetches the task descriptor and invokes the task on the processing unit by (i) providing the address of the first instruction, (ii) specifying the create mask, and (iii) constructing the accum mask for the task. The sequencer determines the next task to be assigned by using information in the task descriptor to predict one of the possible successor tasks (using a static or dynamic prediction scheme). A processing unit independently fetches and executes the instructions of its task (until it encounters an instruction with the stop bit set, which indicates the task is complete). The processing units are connected via a unidirectional ring which is used to forward information (reservations, values, etc.) from one unit to the next [1].

The data cache banks and the associated interconnect (between the data cache banks and the units) are straightforward (except for the scale). Updates of the data cache are not performed speculatively. Instead, additional hardware, known as an *Address Resolution Buffer* or ARB [3-5], is provided to hold speculative memory operations, detect violations of memory dependences, and initiate corrective action as needed[2]. The ARB may be viewed as a collection of the speculative memory operations of the active tasks. The values corresponding to these operations reside in the ARB and update the data cache as their status changes from speculative to non-speculative. In addition to providing storage for speculative operations, the ARB tracks the units which performed the operations with load and store bits. A memory dependence violation is detected by checking these bits (if a load from a successor unit occurred before a store from a predecessor unit, a memory dependence was violated). As the ARB is a finite resource, it may run out of space. If this situation should occur, a simple solution is to free ARB storage by squashing tasks. This strategy guarantees space in the ARB and forward progress. No deadlock problems exists because, in the worst case, all tasks which consume ARB storage may be squashed (the head which does not require ARB storage is not squashed). A less drastic alternative is to stall all processing units but the head. As the head advances, entries are reclaimed and the stall lifted (we are investigating the use of this approach).

Going back to the example of Figure 3, if two symbols being processed concurrently happen to be the same, and a call to the `process` function for the first search updates the memory location corresponding to the symbol, the second search must see the updated memory location. That is, if the unit processing the second symbol loads from the memory location *before* the unit processing the first symbol stores into the memory location, a squash must occur. (A squash does not occur if the dynamic sequence of events is such that the second unit loads from the memory location *after* the first unit stores to the memory location.) Likewise, when a symbol is inserted into the list, subsequent searches must see the updated list. In the same fashion, the cases where later tasks do not see the updated list are detected and the tasks squashed accordingly. Moreover, the storage provided by the ARB is used to rename memory such that multiple function calls can be executed in parallel, yet retain sequential semantics. That is, if multiple calls to `process` are to proceed in parallel, each call requires its own (suitably renamed) stack frame which, as per a sequential execution, reuses the same memory locations.

The microarchitecture illustrated in Figure 1 is just one possible configuration for a multiscalar processor; other microarchitectures are certainly possible. The invariant that has to be preserved is the appearance of a sequential ordering amongst the instructions, with the register and memory values flowing from earlier tasks to later tasks. An alternative microarchitecture might share the functional units (such as the floating point units) between the different processing units. Another possible microarchitecture is one in which the ARB and the data caches are moved across the interconnect to the same side as the processing units. (In this case, the functionality of the ARB and data caches is provided by a collection of temporally inconsistent caches/buffers with memory values forwarded between them on a ring, analogous to the mechanism for registers.) A proper discussion of these alternate microarchitectures is beyond the scope of this paper.

[2] Since the task at the head is the only task that is guaranteed to be non-speculative, memory operations carried out by all units, except the head, are speculative.

3. Distribution of Cycles in Multiscalar Execution

We now take a more detailed look at the multiscalar model by considering the distribution of the available processing unit cycles in multiscalar execution. Recall that our objective is to have each processing unit performing useful computation, with the processing units collectively executing multiple instructions in a given cycle. The best case is to perform as much useful computation per cycle as the processor complex is capable. The best case (of all useful computation) may not be realized because of cycles in which a unit (i) performs non-useful computation, (ii) performs no computation, or (iii) remains idle. Each cycle spent in these categories is a cycle that is lost from the best case.

The non-useful computation cycles represent work that is ultimately squashed; computation may be squashed as a result of the use of (i) an incorrect data value or (ii) an incorrect prediction. The no computation cycles may be attributed to (i) waiting for a value created by an instruction in a predecessor task, (ii) waiting for a value created by an instruction in the same task (for example, a high-latency operation or a cache miss), or (iii) waiting for the task to be retired at the head (because all instructions within the task have executed). The idle cycles account for time in which a processing unit has no assigned task (due for the most part to re-assigning tasks in squash recovery). Below, we discuss several concepts and see the influence on the non-useful and no computation cycles in multiscalar execution. (We do not address the loss due to idle cycles as it amounts to a relatively insignificant portion of the total in most cases.) Although we discuss a concept/issue under one heading, the impact typically spans multiple headings.

3.1. Non-Useful Computation Cycles

Since squashing a particular task means likewise squashing all tasks that follow it, a squash may have a severe impact on the performance of a multiscalar processor. Recall that computation may be squashed as a result of the use of (i) an incorrect value or (ii) an incorrect prediction. To reduce the impact of this squash overhead, we may (i) reduce the chances of a squash by synchronizing data communication or (ii) determine early, before much non-useful computation has been performed, that a squash is inevitable.

3.1.1. Synchronization of Data Communication

The communication of register data values is synchronized as a consequence of the register file mechanism (as intended). On the other hand, the communication of memory data values must be synchronized explicitly. A *memory order squash* occurs if a later task loads from a memory location before an earlier task stores to this same memory location.

Our experience in the programs that we have examined is that such squashes do indeed occur in practice, but rarely are the squashes due to updating an arbitrary memory location. Almost all memory order squashes that we have encountered in our experiments occur due to updates of global scalars and structures, typically file and buffer pointers and counters. (Typically these variables have their address taken, and therefore cannot be register allocated.)

Fortunately, accesses to static global variables are amongst the easiest memory accesses for a compiler to analyze, much easier than accesses to arbitrary heap locations. Once (potentially) offending accesses are recognized, accesses to the memory location can be synchronized to ensure that conflicting loads and stores occur in the proper order.

Such synchronization may be accomplished in a variety of ways. It may possible to create an artificial dependence on a register (to synchronize memory communication with register communication), to delay the load for a given number of cycles (to reduce the probability of it occurring before the store), or to use explicit signal-await synchronization. Note that any synchronization may create inter-task dependences which, as we shall see, can contribute to no computation cycles.

3.1.2. Early Validation of Prediction

The determination of whether a task should be squashed due to an incorrect prediction is normally made at such time as the exit point of the immediately preceding task is known. As one might expect, this point is in most cases at the end of the execution of a task. During this passage of time, many cycles of non-useful computation may have been performed in later tasks.

For example, if loop back is predicted each time for a loop, we may have to wait for all instructions in the last iteration to be executed before we recognize the following iterations are non-useful computation that must be squashed. If an iteration consists of hundreds of instructions, the time taken to determine that no more iterations should be executed may represent many hundreds of cycles of non-useful computation.

To minimize the loss due to these cycles, we may consider validating prediction early. If some computation is performed soon after a task is initiated to determine whether the next task was indeed predicted correctly, the time spent for non-useful computation may be significantly reduced. Returning to the loop example, if the last loop iteration is recognized soon after the iteration begins execution, the next unit may be redirected to the task at the loop exit rather than execute another (non-useful) loop iteration.

Several options exist for validating prediction early. One option is to introduce explicit validate prediction instructions into a task. Another option, directed specifically at loop iterations, which does not require new instructions (but still requires additional instructions as compared to sequential execution), is to change the structure of the (compiled) loop so that the test for loop exit occurs at the beginning of the loop.

3.2. No Computation Cycles

It is important to distinguish between idle cycles and no computation cycles. In the idle cycles case, the processing unit does not perform useful computation because it has no assigned task. In the no computation cycles case, the processing unit does have an assigned task, but it is unable to perform useful computation. Of these lost cycles, some may be an unavoidable characteristic inherent in the sequential code, while others may be a by-product of the task partitioning and scheduling for multiscalar execution.

3.2.1. Intra-Task Dependences

An obvious source of no computation cycles is dependences between the instructions of the same task. As each task is like a small program, and each processing unit is like a uniprocessor, any of the plethora of techniques available to reduce lost cycles in a uniprocessor may be applied to reduce the impact of such cycles. Examples of these techniques include (but need not be limited to) code scheduling, out-of-order execution, and non-blocking caches.

3.2.2. Inter-Task Dependences

A more significant source of no computation cycles in multiscalar execution are dependences between the instructions of different tasks. That is, cycles in which a later task waits for values from an earlier task. If a producing instruction is encountered late and a consuming instruction is encountered early among tasks executing concurrently, the consuming task may stall on the producing task. In such a case, near-sequential execution may result.

Consider our working example. If the induction variable for the outer loop had been updated at the end of the loop (as would normally be the case in code compiled for a sequential execution), then all iterations of the outer loop would be serialized, since the next iteration needs the induction variable early in order to proceed. If, on the other hand, we update and forward the induction variable early in the task, but keep a copy of the induction variable for local use or modify the local use to factor in the update (as we have done in the code of Figure 4), then the critical path through the computation is not unnecessarily aggravated, and the tasks may proceed in parallel.

In our experience with benchmark programs, we have found this sequential outlook to be quite pervasive. The sequential point of view is understandable, since the programmer assumes a sequential machine model. Furthermore, there is no reason to assume a performance improvement is to be gained by making local copies of variables or by making arcane modifications to existing code. Nevertheless, for efficient multiscalar execution, it is crucial to remove such limitations. In many cases, a compiler may have great success (for example, arithmetic induction variables). In other cases, a compiler may have only limited success (for example, memory induction variables). In some cases, these impediments may be unavoidable or require changes to the source program to be overcome.

3.2.3. Load Balancing

In multiscalar execution, since tasks must be retired in order, cycles may be lost if tasks are not of the proper granularity and (roughly) the same size in terms of dynamic instructions. That is, a processing unit which completes a comparatively short task performs no computation while it waits for all predecessor tasks to be retired at the head[3].

---

[3] These no computation cycles may be reduced if we provide a somewhat more complicated implementation of the ‘‘circular queue’’ which connects the units and additional resources to maintain the results of speculative task execution.

A key factor in minimizing cycles lost due to load balancing (and many of the other lost cycles for that matter) is to choose tasks of an appropriate granularity. Flexibility in the choice of the grain size of a task implies that only minimal restrictions be placed on what may be contained in a task. In particular, a task should be free to contain function calls. (In our working example, the appropriate granularity for a task is an iteration of the outer loop, which contains a function call.)

Since a function may have many call sites, we provide differing views on how a function should be executed. From one call site we may want the function to be executed as a collection of tasks. Whereas, from another call site we may want the entire function to be executed as part of a single task. To accommodate such differing views with a single version of the code, a function may be treated as a *suppressed function*, *i.e.*, a function in which all multiscalar-specific annotations are ignored under appropriate circumstances.

4. Comparison of Multiscalar with Other Paradigms

4.1. Conventional Wisdom

The multiscalar paradigm challenges conventional wisdom in ILP processing in several respects. Herein, we examine a number of cases in which the multiscalar approach counters the tenets of conventional wisdom.

Branch prediction accuracy must limit ILP.

The issue at hand is the ability to establish a large and accurate instruction window for ILP extraction. The usual argument supposes that if the average branch prediction accuracy is 90%, then speculating five branches ahead means there is only about a 60% chance that instructions beyond the fifth branch are along the correct dynamic execution path (an 85% accuracy yields less than 45% chance).

A multiscalar processor can speculate across many more than five branches, while still having a very high probability of following the correct dynamic path. In essence, such behavior may be provided by only selectively predicting branches. A multiscalar processor breaks the sequential instruction stream into tasks. Although the tasks may contain internal branches, the sequencer only needs to predict the branches that separate tasks. The branches contained within a task do not have to be predicted (unless they are predicted separately within the processing unit).

In the example of Figure 3, branches in the outer loop delineate the tasks and are predicted (with high accuracy). No branches within the linked list search have to be predicted. In fact, the individual branches that are part of the process of traversing the linked list would likely be predicted not taken because a symbol only matches one element of the list. Nevertheless, the branch for the match will eventually be taken. Suppose we encounter an average of 20 branches (match tests) in traversing the linked list, the execution of an 8-unit multiscalar processor might span 160 conditional branches, yet still be following the correct dynamic path.

The conventional approach, which must sequentially predict all branches as it proceeds, is practically guaranteed to predict wrong eventually (and will never have instructions from more than one list search in progress simultaneously). The multiscalar approach, on the other hand, may overcome this limitation. The ability of a multiscalar processor to selectively bypass branches possibly obviates the need for techniques such as guarded execution, whose net result is also avoiding the prediction of ''bad'' branches (albeit non-loop branches), but at the expense of executing extra instructions [7, 9, 10].

A wide window of pending instructions requires the complexity of concurrently monitoring the issue state of all individual instructions in this window.

In general, instructions from a wide window are selected for execution in parallel and often out-of-order with respect to the sequential program. In a multiscalar implementation, the window can be very wide, yet at any given time only a few instructions need to be inspected for the ability to issue (as few as one for each processing unit). The boundaries of the window of pending instructions can be identified among the active tasks as the first instruction being considered for issue at the head and the last instruction at the tail. As a task may contain a hundred or more dynamic instructions (consider the linked list example in Figure 3), the effective window size can be many hundreds of instructions.

To issue n instructions simultaneously, there must be logic of n^2 complexity to perform dependence cross-checks among the instructions.

That is, issue complexity grows as n^2 to support n-way issue. In a superscalar processor, this observation constrains the capacity of the issue logic. In a multiscalar processor, though, issue logic is distributed to simultaneously fetch and execute multiple instruction streams. Each processing unit issues its instructions in an independent manner. The complexity only consists of multiple copies of relatively simple low-dimension scalar issue logic. The sequencer logic does not have to examine individual instructions as is typically the case in the superscalar approach.

All loads and stores must be identified, and the referenced addresses must be computed, before memory accesses can be re-ordered.

In a conventional implementation, loads and stores are given sequence numbers (or are kept in original sequence) and maintained in a buffer along with the address of the associated memory access. If a load is to be issued, the buffer is checked to ensure that no earlier store to the same address or an unresolved address is pending. If a store is to be issued, the buffer is checked to ensure that no earlier load or store to the same address or an unresolved address is pending. In a multiscalar implementation, loads and stores may be issued independently without knowledge of loads and stores in predecessor or successor tasks.

4.2. Other Paradigms

The superscalar and VLIW approaches, for the most part, follow the conventional wisdom outlined above. A typical superscalar processor fetches the stream of instructions, *examining all instructions as it proceeds* (perhaps multiple instructions are examined at once, but all are examined). Generally, this examination is done to extract and process branch instructions, to identify instruction types so that they may be routed to the proper instruction buffers or reservation stations, and to do some processing to alleviate data dependences, *e.g.*, register renaming [8, 11]. A typical VLIW processor relies on the compiler to perform statically these same functions performed by the superscalar processor dynamically.

In the superscalar approach, it is possible, to generate a fairly accurate window that may be a few branches deep (using a sophisticated dynamic branch predictor), because run-time information is available. Moreover, it is possible to generate a very flexible instruction schedule. For example, it may be possible to allow a load in a callee function to execute in parallel with a store from a caller function. Nevertheless, a superscalar processor has no advance knowledge of the program CFG; it must discover the CFG as it decodes branches. This lack of vision regarding ‘‘what lies ahead’’ and the need to predict every branch limits its ability to create as large or as accurate a window as is possible. Moreover, to extract parallelism from the window requires predominantly centralized resources, including much associative logic, which can be difficult to engineer as the level of ILP increases.

In the VLIW approach, the resulting window may not be very large or may contain inaccuracies arising from static branch prediction, since run-time information is not available to the compiler. Due to this lack of run-time information and the presence of inherent ‘‘boundaries’’ in the program, the ability to move operations in a VLIW processor may be hindered. For example, it may not be possible to provide a static guarantee to allow a load operation in a callee function to execute in parallel with a store operation from a caller function (especially if the callee function is determined dynamically). Furthermore, a VLIW implementation requires a large storage name-space, multiported register files, extensive crossbar interconnects, and stalls if the run-time situation is different from the situation assumed when a code schedule was generated (for example, a cache miss at run-time). Moreover, going from one generation to another may require the problematic re-engineering of program binaries.

In many ways a multiscalar processor is similar to a multiprocessor with very low scheduling overhead[4]. (Both are capable of dispatching large blocks of parallel code.) However, there is a major difference. Whereas a multiprocessor requires a compiler to divide a program into tasks where all dependence relations between tasks are known (or are conservatively provided for) [2], a multiscalar processor *requires no such knowledge* of control and data independence. *If* a compiler can divide a program into tasks that are guaranteed to be independent (for example iterations of a vectorizable loop), of course a multiscalar processor can execute them in parallel. However, the strength in the multiscalar approach lies in executing tasks that are very likely independent or where dependence is relatively low (and therefore ILP exists), *but in the cases for which this information cannot be determined statically* (such as the code of Figure 3).

A multiprocessor with low scheduling overhead, as could be achieved with multiple processors on a chip with a shared cache, is still a multiprocessor. The fundamental automatic parallelization problem is no different from the one computer scientists have struggled with for many years. It may increase the amount of parallelism over conventional parallel processors by differences in scale rather than differences in kind. That is, the lower communication overhead may make some small pieces of code efficient for multiprocessing in more instances than are possible in a conventional multiprocessor. However, new kinds of parallelism are no easier to discover.

A multiscalar processor should also not be confused with a multithreaded processor. In a multithreaded processor, there are multiple threads, or loci of control, which are control independent and (typically) data independent. In contrast, the different ‘‘threads’’ executing on a multiscalar processor are related as different parts of a sequential walk through the same program, and are not control and data independent.

5. Performance Evaluation

5.1. Methodology

All of the results in this paper have been collected on a simulator that faithfully represents a multiscalar processor. The simulator accepts annotated big endian MIPS instruction set binaries (without architected delay slots of any kind) produced by the multiscalar compiler, a modified version of GCC 2.5.8. In order to provide results which reflect reality with as much accuracy as possible, the simulator performs all of the operations of a multiscalar processor and executes all of the program code, except system calls, on a cycle-by-cycle basis. (System calls are handled by trapping to the OS of the simulation host.)

The pipeline structure of a processing unit is a traditional 5 stage pipeline (IF/ID/EX/MEM/WB) which can be configured with in-order/out-of-order and 1-way/2-way issue characteristics. Instructions complete out-of-order and are serviced by a collection of pipelined functional units (1 or 2 simple integer FU, 1 complex integer FU, 1 floating point FU, 1 branch FU, and 1 memory FU) according to the class of the particular instruction with the latencies indicated in Table 1. The unidirectional ring connecting a multiscalar configuration of the processing units imposes a cycle for communication latency between units and matches the ring

Integer	Latency	Float	Latency
Add/Sub	1	SP Add/Sub	2
Shift/Logic	1	SP Multiply	4
Multiply	4	SP Divide	12
Divide	12	DP Add/Sub	2
Mem Store	1	DP Multiply	5
Mem Load	2	DP Divide	18
Branch	1		

Table 1: Functional Unit Latencies.

[4] When compared to a multiprocessor with a low synchronization/scheduling overhead, it is worth noting that the name-space used to synchronize the various units in multiscalar is a common register name-space -- the same register name-space that is used for all computations. In a multiprocessor, we would need separate name-spaces (private registers) for local computation, and (shared registers or main memory) for shared communication, with (possibly explicit) movement of values from one name-space to another. This movement adds overhead.

width to the issue width of the individual units.

All memory requests are handled by a single 4-word split transaction memory bus. Each memory access requires a 10 cycle access latency for the first 4 words and 1 cycle for each additional 4 words. Both loads and stores are non-blocking. In addition, each processing unit is configured with 32 kbytes of direct mapped instruction cache in 64 byte blocks. (An instruction cache access returns 4 words in a hit time of 1 cycle with an addition penalty of 10+3 cycles, plus any bus contention, on a miss.) A crossbar interconnects the units to twice as many interleaved data banks. Each data bank is configured as 8 kbytes of direct mapped data cache in 64 byte blocks with a 256 entry address resolution buffer, for a total of 64 kbytes and 128 kbytes of banked data storage for 4-unit and 8-unit multiscalar processors respectively. (A data cache access returns 1 word in a hit time of 2 cycles and 1 cycle for multiscalar and scalar processors, respectively, with an additional penalty of 10+3 cycles, plus any bus contention, on a miss.)

The sequencer maintains a 1024 entry direct mapped cache of task descriptors. The control flow prediction of the sequencer uses a PAs configuration [12] with 4 targets per prediction and 6 outcome histories. The prediction storage is composed of a first level history table that contains 64 entries of 12 bits each (2 bits for each outcome due to 4 targets) and a set of second level pattern tables that contain 4096 entries of 3 bits each (1 bit target taken/not taken and 2 bits target number). The control flow prediction is supplemented by a 64 entry return address stack.

5.2. Benchmarks

We used the following programs as benchmarks (with inputs other than standard and/or modifications indicated in parentheses): *compress*, *eqntott*, *espresso* (ti.in), *gcc* (integrate.i), *sc* (loada1), and *xlisp* (6 queens) from the SPECint92 suite, *tomcatv* (N=129) from the SPECfp92 suite, *wc* from the GNU textutils1.9 and *cmp* from the GNU diffutils2.6 (two Unix utilities used as benchmarks by the IMPACT group [6], with inputs provided by them), as well as the example from Figure 3 (with an input file of 16 tokens, each appearing 450 times in the file).

Program	Instruction Count		Percent
	Scalar	Multiscalar	Increase
Compress	71.04M	81.21M	14.3%
Eqntott	1077.50M	1237.73M	14.9%
Espresso	526.50M	615.95M	17.0%
Gcc	66.48M	75.31M	13.3%
Sc	409.06M	460.79M	12.6%
Xlisp	46.61M	54.34M	16.6%
Tomcatv	582.22M	590.66M	1.4%
Cmp	0.98M	1.09M	10.9%
Wc	1.22M	1.43M	17.3%
Example	1.05M	1.09M	4.2%

Table 2: Benchmark Instruction Counts.

Table 2 presents the dynamic instruction counts for both scalar and multiscalar execution. (We have only one version of a multiscalar program; the same multiscalar binary is used for all the multiscalar configurations in our experiments.) The extra instructions in a multiscalar program serve to ensure correct execution (such as the use of release instructions) or to enhance performance (such as the creation of local copies of loop induction variables and validating prediction). At present, these instructions unavoidably increase the overall instruction count.

5.3. Results

In Tables 3 and 4 we present the instructions per cycle (IPC) for a scalar execution, the speedups (over the corresponding scalar execution) for 4-unit and 8-unit multiscalar configurations, and the task prediction accuracies. In each case, we report results of the entire execution of the benchmark, not just isolated parts. The results of Table 3 reflect the performance for processing units with in-order 1-way or 2-way issue. Similarly, the results of the Table 4 reflect the performance for processing units with out-of-order 1-way or 2-way issue. The speedups are for a multiscalar processor compared to a scalar processor, in which both use identical processing units. From the data presented in Tables 2, 3, and 4, it is possible to determine the cycle counts in each case. (For example, with 2-way, out-of-order issue processing units, a scalar processor takes 817,845 cycles to execute *Example*, whereas an 8-unit multiscalar processor takes 228,771 cycles.)

In interpreting the results, it is useful to keep a few points in mind. First, Amdahl's law: achieving infinite speedup in only 50% of the code speeds up total performance by only a factor of 2. Second, the IPC of our base scalar configurations is fairly high due to our use of aggressive processing units. Third, we have made no attempt, at this point, to schedule the multiscalar code to tolerate the additional cycle of latency it experiences (as compared to a scalar configuration) for cache hits. Fourth, we have not spent sufficient effort in reducing the additional instructions encountered in multiscalar execution. Finally, we do not give the multiscalar code any ''unfair'' optimization advantages; any optimizations such as loop unrolling are made on both scalar and multiscalar code.

In *compress* all time is spent in a single (big) loop, which contains a complex flow of control within. This loop is bound by a recurrence (getting the index into the hash table) that results in a long critical path through the entire program. The problem is further aggravated by the huge size of the hash table, which results in a high rate of cache misses.

Most (85%) of the instructions in *eqntott* are in the *cmppt* function, which is dominated by a loop. The compiler automatically encompasses the entire loop body into a task, allowing multiple iterations of the loop to execute in parallel.

The top function in *espresso* is *massive_count* (37% of instructions). The *massive_count* function has two main loops. In both cases, the loop body is a task, allowing the multiple iterations to run in parallel. In the first loop, each iteration executes a variable number of instructions (cycles are lost due to load balance). In the second loop (which contains a nested loop), an iteration of outer loop includes all the iterations of the inner loop (in this situation, the task

Program	1-Way Issue Units					2-Way Issue Units				
	Scalar IPC	Multiscalar				Scalar IPC	Multiscalar			
		4-Unit		8-Unit			4-Unit		8-Unit	
		Speedup	Pred	Speedup	Pred		Speedup	Pred	Speedup	Pred
Compress	0.69	1.17	86.8%	1.50	86.1%	0.87	1.04	86.8%	1.34	86.4%
Eqntott	0.83	2.05	94.8%	2.91	94.6%	1.10	1.82	94.8%	2.58	94.6%
Espresso	0.85	1.34	85.9%	1.59	85.9%	1.11	1.22	85.3%	1.41	85.2%
Gcc	0.81	1.02	81.2%	1.08	80.9%	1.04	0.92	81.2%	0.98	80.9%
Sc	0.75	1.36	90.5%	1.68	90.0%	0.94	1.28	90.0%	1.56	89.5%
Xlisp	0.80	0.91	80.6%	0.94	79.5%	1.03	0.86	80.0%	0.88	78.7%
Tomcatv	0.80	3.00	99.2%	4.65	99.2%	0.97	2.71	99.2%	3.96	99.2%
Cmp	0.95	3.23	99.4%	6.24	99.4%	1.32	3.02	99.4%	5.82	99.4%
Wc	0.89	2.37	99.9%	4.33	99.9%	1.09	2.36	99.9%	4.27	99.9%
Example	0.79	2.79	99.9%	3.96	99.9%	1.07	2.43	99.9%	3.47	99.9%

Table 3: In-Order Issue Processing Units.

Program	1-Way Issue Units					2-Way Issue Units				
	Scalar IPC	Multiscalar				Scalar IPC	Multiscalar			
		4-Unit		8-Unit			4-Unit		8-Unit	
		Speedup	Pred	Speedup	Pred		Speedup	Pred	Speedup	Pred
Compress	0.72	1.23	86.7%	1.56	86.0%	0.94	1.07	86.7%	1.33	86.3%
Eqntott	0.84	2.23	94.8%	3.35	94.6%	1.21	1.79	94.8%	2.64	94.5%
Espresso	0.88	1.47	85.9%	1.73	85.8%	1.31	1.12	85.3%	1.25	85.4%
Gcc	0.83	1.06	81.1%	1.13	80.6%	1.15	0.91	81.1%	0.95	80.6%
Sc	0.80	1.42	90.5%	1.75	90.0%	1.10	1.24	90.2%	1.50	90.2%
Xlisp	0.82	0.95	75.6%	1.01	77.1%	1.12	0.85	74.6%	0.90	76.5%
Tomcatv	0.96	2.92	99.2%	4.17	99.2%	1.43	2.16	99.2%	2.93	99.2%
Cmp	0.95	3.24	99.2%	6.28	99.1%	1.68	2.76	99.2%	5.30	99.2%
Wc	0.89	2.37	99.9%	4.34	99.9%	1.13	2.34	99.9%	4.26	99.9%
Example	0.86	3.27	99.9%	4.86	99.9%	1.28	2.41	99.9%	3.57	99.9%

Table 4: Out-Of-Order Issue Processing Units.

partitioning needed a manual hint to select this granularity).

Both *gcc* and *xlisp* distribute execution time uniformly across a great deal of code. These are also the programs that we have, to date, spent the least amount of time analyzing. In both these cases, for the task partitioning that we use currently, squashes (both prediction and memory order) result in near-sequential execution of the important tasks. Accordingly, the overheads in our multiscalar execution (extra instructions and extra cache hit latency) result in a slow down in some cases. (Incidentally, the instruction count is slightly lower than what is typically observed because we unroll the *memset* and *memcpy* functions.) For *gcc* our experience to date suggests that parallelism, which may be exploited by multiscalar, exists; we are less confident about *xlisp* at this point.

In *sc*, the dominant user routine is *RealEvalAll*, though it only accounts for less than 12% of the total instructions. *RealEvalAll* contains a two-level nested loop that makes a call to *RealEvalOne* for appropriate cells of the spreadsheet. *RealEvalOne* further calls *eval* which is a recursive function to evaluate a cell. The body of the inner loop of *RealEvalAll* is a task with the call to *RealEvalOne* suppressed manually. The loop in *RealEvalAll* visits every cell of the spreadsheet. If a cell is not empty, *RealEvalOne* is called to evaluate it, else no action is taken at the cell. Since *RealEvalOne* executes for hundreds of cycles, the load imbalance between the work at each cell is enormous. Accordingly, we restructured the *RealEvalOne* loop to build a work list of the cells to be evaluated and to call *RealEvalOne* for each of the cells on the work list.

For *tomcatv* nearly all time is spent in a loop whose iterations are independent. Accordingly, we achieve good speedup for 4-unit and 8-unit multiscalar processors. The higher-issue configurations are stymied because of the contention on the cache to memory bus.

The programs *cmp* and *wc* are straightforward, with each spending almost all its time in a loop. The loops, however, contain an inner loop (the loop in *wc* also contains a switch statement). In these cases, the performance loss may be attributed mainly to cycles lost due to branches and loads inside each task (intra-task dependences).

Our *example* spends 80% of its time in the code shown in Figure 3, performing the symbol fetch, match, and process or add sequence. The remaining time is spent in fetching the data from the input file into the buffer. Since the iterations of the outer loop are mostly independent (dynamically), we attain excellent speedups. Interestingly, other known ILP paradigms such as superscalar and VLIW are unlikely to extract any meaningful parallelism, in an efficient manner, for this example.

6. Summary and Conclusions

This paper presented the multiscalar processing paradigm, a new paradigm for exploiting fine-grain, or instruction-level parallelism. A multiscalar processor uses a combination of hardware and software to extract ILP from ordinary programs. It does so by dividing the program control flow graph (CFG) into tasks, and stepping through the CFG speculatively, taking large steps, a task at a time, without pausing to inspect the contents of a task. The tasks are distributed to a collection of processing units, each of which fetches and executes the instructions in its task. Collectively, this processor complex uses multiple program counters to sequence through different parts of the program CFG simultaneously, resulting in multiple instructions being executed in a cycle.

We described the philosophy of the multiscalar paradigm, the structure of multiscalar programs, and the hardware architecture of a multiscalar processor. We also discussed several issues related to the performance of a multiscalar processor, and compared the multiscalar paradigm with other ILP processing paradigms. Finally, we carried out a performance evaluation of several multiscalar configurations on an ensemble of well-known benchmarks.

The performance results presented in this paper, in our opinion, only hint at the possibilities of the multiscalar approach. As we investigate the dynamics of multiscalar execution, we continue to evolve the compiler and to better understand its interaction with the hardware. At present, we optimistically view performance impediments as problems for which we have not yet developed solutions. Our expectation is that with improved software support, and more streamlined hardware, multiscalar processors will be able to extract levels of ILP that are far beyond the capabilities of existing paradigms. (We plan to make updated results available on the multiscalar WWW page: URL http://www.cs.wisc.edu/~mscalar.)

Acknowledgements

This work was supported in part by NSF grant CCR-9303030 and by ONR grant N00014-93-1-0465. We would like to thank Jim Smith for his contributions to the multiscalar project in general, and this paper in particular.

References

[1] S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, ''The Anatomy of the Register File in a Multiscalar Processor,'' *Proc. MICRO-27*, pp. 181-190, December 1994.

[2] D. K. Chen, H. M. Su, and P. C. Yew, ''The Impact of Synchronization and Granularity on Parallel Systems,'' *Proc. 17th Annual International Symposium on Computer Architecture*, pp. 239-248, May 1990.

[3] M. Franklin and G. S. Sohi, ''ARB: A Hardware Mechanism for Dynamic Memory Disambiguation,'' *submitted to IEEE Transactions on Computers*.

[4] M. Franklin and G. S. Sohi, ''The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism,'' in *Proc. 19th Annual Symposium on Computer Architecture*, Queensland, Australia, pp. 58-67, May 1992.

[5] M. Franklin, ''The Multiscalar Architecture,'' Ph. D. Thesis, Computer Sciences Technical Report #1196, University of Wisconsin-Madison, Madison, WI 53706, November 1993.

[6] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. W. Hwu, ''Superblock Formation Using Static Program Analysis,'' *Proc. MICRO-26*, pp. 247-255, December 1993.

[7] P. Y.-T. Hsu and E. S. Davidson, ''Highly Concurrent Scalar Processing,'' *Proc. 13th Annual Symposium on Computer Architecture*, pp. 386-395, June 1986.

[8] R. M. Keller, ''Look-Ahead Processors,'' *ACM Computing Surveys*, vol. 7, pp. 66-72, December 1975.

[9] S. A. Mahlke, D. C. Liu, W. Y. Chen, R. E. Hank, and R. A. Bringmann, ''Effective Compiler Support for Predicated Execution Using the Hyperblock,'' in *MICRO-25*, Portland, Oregon, pp. 45-54, December 1992.

[10] D. N. Pnevmatikatos and G. S. Sohi, ''Guarded Execution and Branch Prediction in Dynamic ILP Processors,'' in *Proc. 21th Annual International Symposium on Computer Architecture*, Chicago, Illinois, pp. 120-129, April 1994.

[11] G. S. Tjaden and M. J. Flynn, ''Detection and Parallel Execution of Independent Instructions,'' *IEEE Transactions on Computers*, vol. C-19, pp. 889-895, October 1970.

[12] T.-Y. Yeh and Y. N. Patt, ''A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History,'' in *Proc. 20th Annual International Symposium on Computer Architecture*, San Diego, California, pp. 257-266, May 1993.