To appear in the 31st International Symposium on Microarchitecture, Dec. 1998

Task Selection for a Multiscalar Processor

T. N. Vijaykumar
vijay@ecn.purdue.edu
School of Electrical and Computer Engineering
Purdue University

Gurindar S. Sohi
sohi@cs.wisc.edu
Computer Sciences Department
University of Wisconsin-Madison

Abstract

The Multiscalar architecture advocates a distributed processor organization and task-level speculation to exploit high degrees of instruction level parallelism (ILP) in sequential programs without impeding improvements in clock speeds. The main goal of this paper is to understand the key implications of the architectural features of distributed processor organization and task-level speculation for compiler task selection from the point of view of performance. We identify the fundamental performance issues to be: control flow speculation, data communication, data dependence speculation, load imbalance, and task overhead. We show that these issues are intimately related to a few key characteristics of tasks: task size, inter-task control flow, and inter-task data dependence. We describe compiler heuristics to select tasks with favorable characteristics. We report experimental results to show that the heuristics are successful in boosting overall performance by establishing larger ILP windows.

1. Introduction

Modern microprocessors achieve high performance by exploiting instruction level parallelism (ILP) in sequential programs. They establish a large dynamic window of instructions and employ wide-issue organizations to extract ILP and execute multiple instructions simultaneously. Larger windows enable more dynamic instructions to be examined, which leads to the identification of more independent instructions that can be executed by wider processors. However, large centralized hardware structures for larger windows and wider processors may be harder to engineer at high clock speeds due to quadratic wire delays, limiting overall performance (e.g., the DEC Alpha 21264 already implements a two-cluster pipeline because bypassing across a single larger cluster would not fit within a cycle). The Multiscalar architecture [5] [6] [14] advocates a distributed processor organization to avail of the advantages of large windows and wide-issue pipeline without impeding improvements in clock speeds. The key idea is to split one large window into multiple smaller windows and one wide-issue processing unit (PU) into multiple narrow-issue processing units connected together.

Unless key performance issues are understood, smaller distributed designs may not always perform better than larger centralized designs, despite clock speed advantages. The choice of tasks is pivotal to achieve high performance. While a good task selection may result in the program partitioned into completely independent tasks leading to high performance improvements, a poor task selection may lead to the program partitioned into dependent tasks resulting in performance worse than that of a single processing unit, due to overhead resulting from distributing hardware resources.

The main goal of this paper is to understand the implications of the architectural features of distributed processor organization and task-level speculation for compiler task selection from the point of view of performance. We identify the fundamental performance issues to be: control flow speculation, data communication, data dependence speculation, load imbalance, and task overhead (Section 2). We show that these issues are intimately related to a few key characteristics of tasks: task size, inter-task control flow, and inter-task data dependence (Section 2). Task size primarily affects load imbalance and overhead, inter-task control flow influences control speculation, and inter-task data dependence impacts data communication and data dependence speculation. These issues, which do not exist for centralized microarchitectures that do not perform task-level speculation (e.g., superscalar) have not been studied before in the context of sequential programs and microarchitectures, but are germane to several recent proposals for distributed microarchitectures employing some form of task level speculation, including the MIT RAW [19], Stanford Hydra [12], CMU STAMPede [16], and Minnesota Superthreaded architecture [17]. In Section 3, we describe our compiler heuristics to select tasks with favorable characteristics. In Section 4, we analyze the effects of the various heuristics on overall performance and present measurements of the key task characteristics. We draw some conclusions in Section 5.

2. Overview of a Multiscalar processor

We begin with a description of the execution model of a Multiscalar processor in the abstract but in enough detail to pinpoint problems. We define tasks and then follow the time line of the execution of a task to identify different sources of performance loss. We associate each such phase with a specific task characteristic to motivate the heuristics to select tasks with favorable characteristics.

2.1. Execution model

In a Multiscalar processor, sequential programs are partitioned into sequential (not necessarily independent) **tasks.** Figure 1 illustrates the Multiscalar execution model. Figure 1 shows a static program partitioned into three tasks and three points of search in the dynamic stream with three corresponding windows. Execution proceeds by assigning tasks to PUs. After assigning a task for execution, one of the possible successors of the task is predicted to be the next task, similar to branch prediction employed by superscalar machines, i.e., **control flow speculation** is used. Since the tasks are derived from a sequential program and are predicted in the original program order, the total order among the tasks is unambiguous. The predicted task speculatively executes on a PU using its private resources unless it needs values computed by another task executing on a different PU. Intra-task dependences are handled by the processing units, similar to superscalar processors. In the case of inter-task register data dependences, a producer task **communicates** the required value to the consumer task when it has been computed [3]. In the case of inter-task memory data dependences, **memory dependence speculation** is employed; a task begins by speculating that it does not depend on any previous task for memory values and executes loads from the specified addresses. If the speculation is incorrect (i.e., a previous task performs a store to the same address as a load performed by this task), a memory dependence violation is detected by the Address Resolution Buffer (ARB) [7] and the offending task that performed the load (and all its successor tasks) is **squashed**. In a real implementation, it may be difficult to isolate tasks that are dependent on the offending task and squash only those. If control flow speculation is incorrect (i.e., one of the tasks assigned for execution was incorrect), the incorrect task (and all its successor tasks) is squashed similar to a branch misprediction in superscalar machines. Since tasks execute speculatively, the state (register and memory) produced by it is buffered and cannot update architectural state. When a task completes and all speculations (control flow and memory dependence) have been resolved to be correct, the task is retired (i.e., its speculative state is promoted to architectural state). Tasks are retired in the original program order to maintain sequential program semantics; a task is retired only after its predecessor task has been retired.

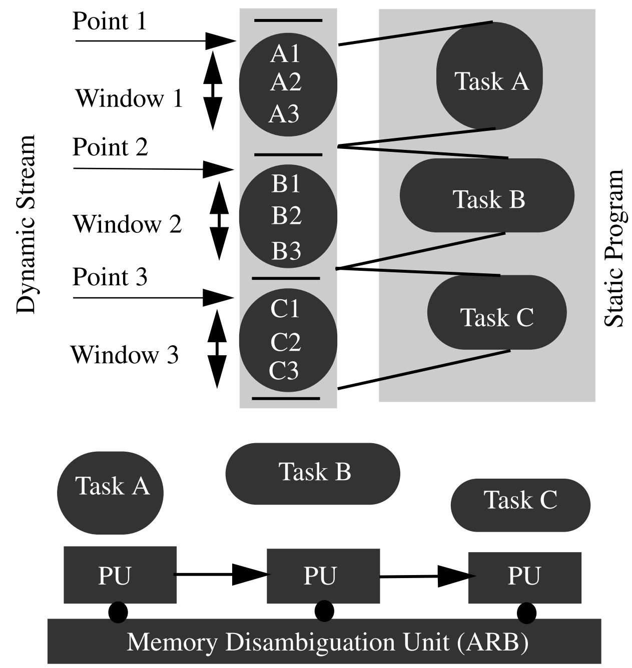

Figure 1: Abstraction of a Multiscalar processor.

2.2. Definition

A Multiscalar task may comprise a basic block, multiple basic blocks, loop bodies, entire loops, or entire function invocations. Statically, a task is defined to be a connected, single-entry subgraph of the static control flow graph (CFG) of a sequential program. Dynamically, a task corresponds to a contiguous fragment of the dynamic instruction stream that may be entered only at the first instruction of the fragment. Since the compiler does not have access to dynamic control flow paths, it treats a set of static control flow paths connected together in a subgraph of the CFG as a task. Thus, a static task contains computation that is a superset of the computation performed by each dynamic invocation of the task. Although inexact, this definition allows the compiler to conveniently perform task selection.

2.3. Time line of a task

We add timing information to the functional description of execution of tasks to account for execution time of tasks. When a task is assigned for execution, two possibilities arise: (1) the task completes and is retired, or (2) an incorrect speculation (control flow or memory dependence) occurs and the task is squashed.

2.3.1. Scenario 1: task is retired

When a task is assigned for execution, it begins by fetching instructions from the start PC and filling the pipeline of the PU with instructions. The time associated with filling the pipeline is classified as **task start overhead**. An instruction executes after its input operands are available, similar to a superscalar machine. If an input operand is not available, then the instruction waits until the value is available. If the instruction waits for a value to be produced and communicated by another task, then the associated wait time is classified as **inter-task data communication delay**. If the instruction waits for a value to be produced by a previous instruction in the same task, then the associated wait time is classified as **intra-task data dependence delay**. As soon as the required value is available, execution proceeds and eventually the task ends. After completion, the task waits until the previous task retires; the associated wait time is classified as **load imbalance**. When the task is allowed to retire, it commits its speculative state to architectural storage; the associated time is classified as **task end overhead**. Figure 2(a) illustrates the various phases of scenario 1.

2.3.2. Scenario 2: task is squashed

When a task is assigned for execution, it proceeds as explained above until the control-flow speculation is resolved. If either the task itself or one of its predecessor task is detected to have misspeculated, the task is squashed and a new task is assigned to the PU. The entire time since the start of the task, irrespective of whether the task was waiting for values or executing instructions, is classified as **control flow misspeculation penalty** or **memory dependence misspeculation penalty**, as the case may be. Since a misspeculation may cause several tasks to be squashed (the offending task and all its successors), the misspeculation is associated with the sum of all the individual penalties of each of the squashed tasks. Figure 2 (b) illustrates the various phases of scenario 2.

2.4. Performance and tasks

From the discussion of the task execution time line, the major categories of performance degradation are: control flow misspeculation, inter-task data communication, memory dependence misspeculation, load imbalance and task overhead. We now relate the performance issues to concrete task characteristics: task size, inter-task control flow, and inter-task data dependence.

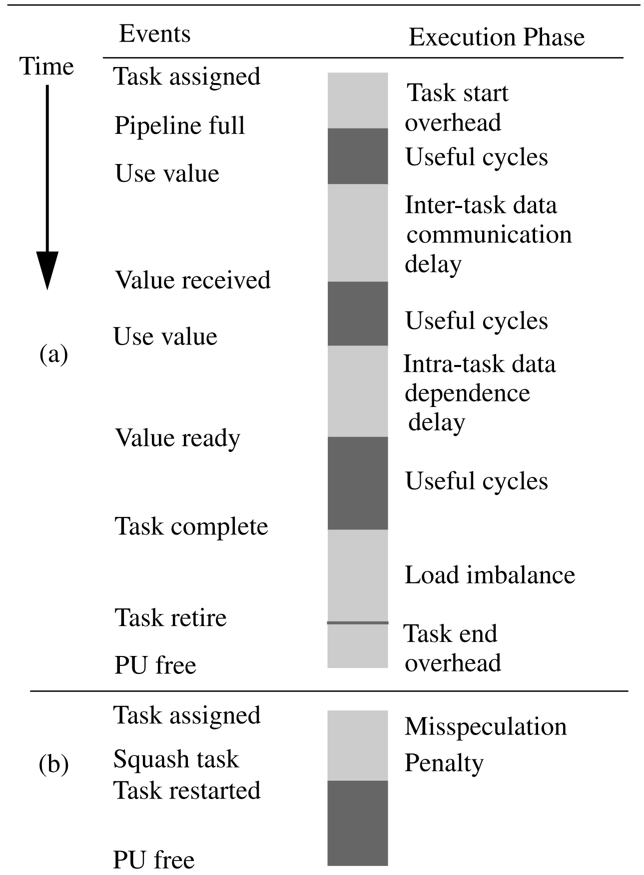

Figure 2: Time line of the execution of a task.

2.4.1. Task size

Small tasks do not expose adequate parallelism and incur high overhead. If tasks contain a large number of dynamic instructions, then several problems arise: (1) Large tasks increase both the number of misspeculations and misspeculation penalty. Speculating over a large number of memory operations usually results in a large likelihood of misspeculating a true data dependence and discarding a large amount of work. (2) Large tasks may cause the ARB to overflow causing the task to stall until speculation is resolved. (3) Large tasks may result in a loss of opportunity because narrow PUs cannot put large amounts of intra-task parallelism to use. Load imbalance causes performance loss similar to large-scale, parallel machines [10]. Large variations in the amount of computation of adjacent tasks causes load imbalance resulting in successor tasks waiting for predecessor task to complete and retire. There are two kinds of overhead associated with tasks: (1) task start overhead, and (2) task end overhead. Task start overhead is primarily caused by pipeline filling at the start of every task, similar to the problem of short vectors in a vector machine. At the end of a task, the speculative state of the task is committed to the architectural storage; if this involves actual movement of data (as opposed to just tagging of data as committed without physical movement) extra cycles are spent.

2.4.2. Inter-task control flow

Control flow edges that are exposed during task selection, giving rise to inter-task control flow, affect control flow speculation by the hardware. The resolution of control flow from one task to the next can be done typically only at the end of the task because the branch instructions that change control flow from one task to next may be encountered only at the end of the task. The result of this "late" resolution is that control flow misspeculation penalties are of the order of the execution time of tasks, which may be many cycles. Both the number of successors and the kind (predictable or not) of successors are key factors. Since a task is assigned for execution via control flow speculation, it is important that tasks have at most as many successors as can be tracked by the hardware prediction tables. These hardware tables are built to track some fixed number (N) of successors. If tasks are selected to have more than N successors, dynamic control flow speculation accuracy decreases, leading to loss of performance. Control flow misspeculation depends on the predictability of the successors and the position of the control flow dependences that are exposed by task selection (dynamically encountered/executed early or late during the execution of the task).

2.4.3. Inter-task data dependence

Data dependence edges that are included within tasks do not cause any data communication delay or dependence misspeculation. Data dependence edges that are exposed during task selection, giving rise to inter-task data dependence, affect data communication and data dependence speculation. Inter-task data dependence delay and data dependence misspeculation cost is determined by the position in the task (dynamically encountered/executed early or late during the execution of the task) of the data dependences that are exposed by task selection. The interval between the instant when a task begins execution and the instant when a producer or consumer instruction executes depends on (1) whether the instruction depends on values from predecessor tasks and if so, the time when the value is produced and communicated and (2) the number of other instructions of the task that are ahead of the instruction as per program order, since instructions in a task are executed on a PU in the usual uniprocessor style of execution. A producer instruction, therefore, may execute much later after the task begins execution if a required value is available later or if many other instructions precede the instruction. Thus, a data dependence may get aggravated to a long delay if split across large tasks. If an inter-task data dependence is misspeculated (instead of waiting) then similar performance loss is incurred. If the producer is executed at the end of its task and the consumer is executed at the beginning of its task, then the squash penalty incurred is proportional to the execution time of the producer task.

3. Selecting tasks

The guiding principle used to select tasks is that control and data dependent computation is grouped into a task so that communication or misspeculation are minimized. Data dependences, control dependences, load imbalance, and task overhead often impose conflicting requirements. Sarkar showed that given the communication costs and amount of work associated with each function invocation, partitioning simple functional programs into non-speculative tasks to optimize the execution-time on a multiprocessor is NP-Complete [13]. Due to the intractable nature of obtaining optimal tasks, we rely on heuristics to approach the problem. This section describes how our task selection heuristics produce tasks with favorable characteristics. However, before going into the description, it may be helpful to summarize the observations that led to our choice of task selection strategies.

We started with a basic process of CFG traversal to perform task selection. To this basic process, we added the heuristics, which are based on the characteristics described earlier, to steer the traversal. The heuristics were implemented in a progression starting with tasks containing a single basic block. To alleviate the performance problems caused by the small size of basic block tasks, multiple basic blocks are included within tasks. But tasks containing multiple basic blocks often incurred excessive inter-task control flow misspeculations. To mitigate control flow misspeculations, the number of successors of a task were controlled through the control flow heuristic. Even though control flow speculation improved, inter-task data dependences were aggravated, resulting in performance loss. To reduce this loss, data dependences were included within tasks through the data dependence heuristic. Profiling was used to integrate all of the heuristics together.

3.1. Basic task selection process

Task selection proceeds by traversing the CFG of the application starting at the root of the CFG. Basic blocks are included in a task by progressively examining whether successors of the basic blocks that have already been included in the task may also be added to the task. The heuristics are incorporated in the selection process via decision-making functions that determine whether a particular basic block should be included in a task or not. If the heuristics terminate a control flow path by not including a basic block, then another task is grown starting at that basic block. Figure 3 shows the heuristics in pseudo-code. *task_selection()* is the top level driver for the process.

3.2. Task size heuristic

Demarcating basic blocks as tasks is easy for the compiler because basic blocks are already identified. Tasks so obtained are called **basic block tasks**. No special decision-making functions are required to generate basic block tasks. Since basic block tasks are too small to expose enough parallelism, multiple basic blocks are assembled into one task. But, generating tasks comprising arbitrarily many basic blocks may lead to the problem of large tasks. Apart from including many basic blocks within tasks, including entire loops and function invocations within tasks also may lead to large tasks. Terminating tasks at function invocations as well as entry and exit of loops naturally limits task size. Most modular applications tend to have function calls, which naturally prevent large tasks. But frequent function invocations with little computation between them and loops with small loop bodies may lead to small tasks. Loops can be unrolled so that multiple iterations of short loops can be included to increase the size of short loop-body tasks; short function invocations can be inlined or included entirely within tasks to avoid small tasks.

task_size_heuristics() in Figure 3 shows the pseudo-code for our implementation. Calls to functions that contain fewer than CALL_THRESH (set to 30) dynamic instructions are included entirely within a task. We chose to include entire calls instead of inlining because inlining may cause code-bloat. CALL_THRESH was set to 30 to keep task overhead to around 6% of task execution time (assuming task overhead of 2 cycles and each instruction takes one cycle). Loop bodies containing fewer than LOOP_THRESH (set to 30) static instructions are unrolled to expand to LOOP_THRESH instructions. Entry into loops, exit out of loops and function calls (with more than 30 instructions) always terminate tasks.

3.3. Control flow heuristic

If multiple basic blocks are included within a task, then the number of successors of each task needs to be controlled. Tasks so obtained are called **control flow tasks**. To control the number of successors of tasks while employing the task size heuristic, basic blocks and control flow edges are categorized as **terminal** or **non-terminal**. If a basic block is terminal, then none of its successors (in the CFG) are included in the task containing the basic block. From Section 3.2, loop back edges and edges that lead into a loop, and basic blocks that end in a function call or a function return are marked as terminal. *is_a_terminal_node*() and *is_a_terminal_edge*() in Figure 3 show the pseudo-code. Terminal edges are not included within a task. Non-terminal edges may or may not be included within a task, depending upon the heuristics.

dependence_task() in Figure 3 shows how the number of successors of a task is controlled. *dependence_task()* explores one basic block per invocation and queues the children of the basic block under consideration for further exploration. During the CFG traversal, if any terminal edges or terminal basic blocks are encountered, then the path is not explored any further, and the basic blocks that terminate the path are marked as successors. *dependence_task()* explores one basic block per invocation and queues the children of the basic block under consideration for further exploration. In order to ensure that tasks have at most N successors, the number of successors is tracked when basic blocks are included within tasks; the largest collection of basic blocks that correspond to at most N successors called the **feasible task** is also tracked. After a basic block is added to the potential task, if the resulting number of successors is at most N, then the basic block is added to the feasible task. By taking advantage of reconverging control flow paths, tasks are made larger without necessarily increasing the number of successors. But during the traversal, it is not known a priori which paths reconverge and which do not. The control flow heuristic uses a greedy approach; the traversal continues to explore control flow paths even if the number of successors exceeds the allowed limit. When all the control flow paths are terminated, the feasible task so obtained demarcates the task.

There are a myriad of techniques to alleviate the problems caused by control flow for scheduling of superscalar code, such as trace scheduling [4], predication [8], and if-conversion [2]. The key point for Multiscalar is that as long as control flow is included within tasks the primary problem of mispredictions is alleviated. Techniques like predication can be employed to improve the heuristics but are not explored here because such techniques need either extra hardware support (predication) or may introduce bookkeeping overhead (trace scheduling). Intra-task control flow may cause performance loss due to delay of register communication. This secondary problem can be alleviated by scheduling using the previously mentioned techniques. For loops, we move the induction variable increments to the top of the loops so that later iterations get the values of the induction variables from earlier iterations without any delay. Register communication scheduling is not discussed in this paper; details are available in [18].

3.4. Data dependence heuristic

The key problem with a data dependence is that if the producer is encountered late and the consumer is encoun-

```
task_selection() {
   task_size_heuristic();
   identify_data_dependences();
   dep_list = sort_datadep_by_freq();
   for each (u,v) in dep_list
         for each t = including_task of u
               expand_task(u, t, (u,v));
   if (not_in_any_task(u))
         expand_task(u, new_task(u), (u,v));
}
task_size_heuristic() {
   for each loop l
         if (loop_size(l) < LOOP_THRESH)
               unroll_loop(l);
   for each block blk ending in a call f
         if (#instructions(f) < CALL_THRESH)
               mark_for_inclusion(blk);
}
is_a_terminal_node(blk) {
   return ((does_not_end_in_call(blk) ||
         !marked_for_inclusion(blk))
         && not_a_loop_end(blk)
         && not_a_loop_head(blk));
}
expand_task(blk, task, dep_edge) {
    explore_q = task->explore_q;
    root = task->root;
    while (not_empty(explore_q))
          dependence_task(blk, root, dep_edge);
}
dependence_task(blk, root, dep_edge) {
    if (! is_a_terminal_node(blk) {
          for each child ch of blk
                if (! is_a_terminal_edge(blk, ch) {
                      if (codependent(ch, dep_edge))
                            add_explore_q(ch);
                      t = adjust_targets(root, blk, ch);
                      if (t < N)
                            feasible_task(root, blk, ch);
                } else
                      add_to_task_q(ch);
    } else
          for each child ch of blk
                add_to_task_q(ch);
}
is_a_terminal_edge(blk, ch) {
   return(dfs_num(blk) < dfs_num(ch));
}
```

Figure 3: Task selection heuristics.

tered early, then many cycles may be wasted waiting for the value to be communicated. The main goal of data dependence driven task selection is that for a given data dependence extending across several basic blocks, either the dependence is included within a task or it is exposed such that the resulting communication does not cause stalls. Tasks so obtained are called **data dependence tasks**.

During the selection of a task, the data dependence heuristic steers the exploration of control flow paths to those basic blocks that are dependent on the basic blocks that have been included in the task. The Control flow heuristic includes basic blocks in tasks regardless of whether they are dependent on other basic blocks contained in the task. The Data dependence heuristic, instead, includes a basic block only if it is dependent on other basic blocks included in the task. Thus, the data dependence heuristic explores only those control flow paths that lead to dependent basic blocks and terminate the other paths. There are several impediments to including a data dependence within a task: (1) many memory dependences are unknown or ambiguous at compile-time and (2) including a data dependence within a task may result in a task with more successors than desired.

There are many data dependence detection techniques for memory dependences through memory disambiguation schemes [3] [20]. These techniques work well for programs that do not employ intricate pointers. Due to the prevalence of pointers in most of our benchmarks, we rely on the memory dependence synchronization mechanism [11] to avoid excessive squashing and the ARB to ensure correctness. But register dependences are identified and specified entirely by the compiler using traditional def-use dataflow equations [1]. Space constraints do not permit us to list the equations here.

dependence_task() in Figure 3 integrates the data dependence heuristic with the control flow heuristic. For each data def-use dependence, we try to include the dependence within a task, without exceeding the limit on the number of successors. If the producer is already included in a task, then that task is expanded in an attempt to include the dependence; otherwise, a new task is started at the producer. In general, if the producer and the consumer are not in adjacent basic blocks in the control flow graph, then the basic blocks in all the control flow paths from the producer to the consumer also have to be included. The set of basic blocks in all the control flow paths from the producer to the consumer is called the **codependent set** of the dependence. Codependent sets are identified by the dataflow equations that determine the def-use chains. *codependent()* determines whether a basic block is in the codependence set of a data dependence edge or not. The heuristic attempts to include a register dependence within a task by steering the basic traversal to include the codependent set. For the cases where a dependence cannot be included due to exceeding the limit on the number of successors, the heuristic avoids poor scheduling of the resultant communication. If the producer is not dependent on any other computation, then the heuristic starts a task at the producer enabling early execution of the producer. If more than one dependence is taken into consideration, then including one dependence may exclude another because inclusion of a certain dependence may result in some control flow paths to be terminated and inclusion of another dependence may require some of the previously terminated paths to be not terminated. A simple solution to this difficulty is to prioritize the dependences using the execution frequency of the dependences, obtained by profiling. More details on the heuristics are in [18].

3.5. Tasks selected by the heuristics

Figure 4 illustrates task partitions that may result when a data dependence edge is considered. Figure 4(a1) shows a part of the CFG of a program including a data dependence edge from the top basic block to the bottom basic block. ➤ indicates a control flow edge and ⟶ indicates a data dependence edge. For this example, let us assume that the number of hardware targets is 4. Since the control flow heuristic does not take data dependences into consideration, the dependence is split between Task1 and Task2. Figure 4(a2) shows a task that includes the data dependence edge by including all the basic blocks in the control flow path from the producer basic block to the consumer basic block. Figure 4(b1) shows a task partition obtained by the control flow heuristic. Since the control flow heuristic does not take data dependences into consideration, the producer of the data dependence is included at the end of Task1 and the consumer is included at the beginning of Task3, aggravating data dependence delay. Figure 4(b2) shows a task partition obtained by the data dependence heuristic consisting of two tasks in which the resultant inter-task communication is scheduled favorably; the producer instruction is placed early in its task at the first basic block of the task and the consumer instruction is placed late in its task at the last basic block of the task.

4. Experimental evaluation

The heuristics described in the preceding sections have been implemented in the Gnu C Compiler, gcc. SPEC95 benchmark [15] source files are input to the compiler which produces executables. The binary generated by the compiler is executed by a simulator which faithfully captures the behavior of a Multiscalar processor on a cycle per cycle basis by simulating all instructions except for system calls.

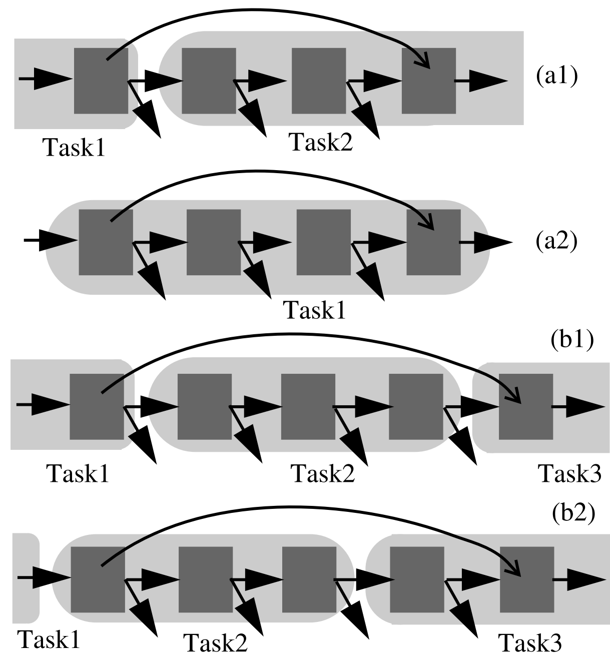

Figure 4: Tasks and data dependence edges.

4.1. Overview of experiments

In this section, we describe the quantities measured in the experiments to analyze performance issues and demonstrate the impact of the compiler heuristics. In order to determine the effectiveness of the compiler heuristics, we measure performance of basic block tasks, control flow tasks, and data dependence tasks.

For each of the task characteristics: task size, inter-task control flow, and inter-task data dependence, we measure a metric that closely correlates to performance. We measure the average number of dynamic instructions per task. For inter-task control flow, we measure prediction accuracies to estimate the magnitude of control flow misspeculation. By studying the nature of the dynamic window established by Multiscalar organizations, we can estimate the amount of parallelism that the machine can exploit. For superscalar processors, the average window size is a good metric of quality of the dynamic window. Since a Multiscalar processor does not establish a single continuous window, we extend window size to another metric. For a Multiscalar processor, the total number of dynamic instructions that belong to all the tasks in execution simultaneously called the **window span** is the equivalent of the superscalar window as far as potential exploitable parallelism is concerned. We present the average window span for the benchmarks.

4.2. Simulation parameters

The simulator models details of the processing units, the sequencer, the control flow prediction hardware, the register communication ring, and the memory hierarchy consisting of the ARB, L1 data and instruction cache, L2 cache, main memory, and interconnection between the PUs and the ARB and the L1 caches, the L1 caches and the L2 cache, as well as the system bus connecting the L2 caches and the main memory. Both access latencies and bandwidths are modeled at each of these components and the interconnects. The PUs are configured to use 2-way issue, 16-entry reorder buffer, and 8-entry issue list with two integer, one floating point, one branch, and one memory functional units. The intra-task prediction uses *gshare* with 16-bit history, and 64K-entry table of 2-bit counters. The inter-task prediction uses a path-based scheme [9] with 16-bit history, 64K-entry table of 2-bit counters and 2-bit target numbers. The register communication ring can carry 2 values per cycle and bypass values in the same cycle between adjacent PUs. The L1 I-cache characteristics are: 64KB (4PU)/ 128KB (8PU), 2-way associative, 32 byte blocks, 1 cycle hit, interleaved as many banks as the number of PUs, lock-up free, fully pipelined, and augmented with a 32KB, 2-way associative task cache. The L1 D-cache characteristics are: 64KB (4PU)/128KB (8PU), 2-way associative, 32 byte blocks, 1 cycle hit, interleaved as many banks as the number of PUs, lock-up free, fully pipelined, and augmented with a 32KB, 2-way associative task cache. The ARB characteristics are: 32 entries/PU, 32 x #PU bytes/entry, 4KB (4PU)/8KB (8PU), fully associative, 2 cycle hit, interleaved as many banks as the number of PUs, lock-up free, pipelined, and augmented with a 256-entry memory synchronization table [11]. The L2 cache is 4MB, and 2-way associative with 12 cycle hit latency and 16 bytes per cycle transfer. Finally, the main memory is infinite capacity with 58 cycle latency, and 8 bytes per cycle transfer.

All of the binaries for the experiments are generated with the highest level of gcc 2.7.2 optimizations. For Fortran programs, we use f2c and then compile the C code with our compiler. Multiscalar-specific optimizations including task selection, loop restructuring, dead register analysis for register communication, and register communication scheduling and generation are also used [18]. The compiler uses basic block frequency, obtained via dynamic profiling, for register communication scheduling and task selection. Profiling was done using the inputs specified by the SPEC95 suite.

4.3. Experiments and Results

4.3.1. Effectiveness of the heuristics

Figure 5 shows the improvements in IPC using the control flow heuristic, the data dependence heuristic, and the task size heuristic for out-of-order and in-order PUs executing the integer benchmarks and floating point benchmarks over the base case of basic block tasks.

The compiler heuristics (control flow, data dependence and task size together) are effective in capturing parallelism beyond basic block tasks. Using out-of-order PUs, the integer benchmarks improved by 19-38% and 25-39% on 4 and 8 PUs, respectively, over basic block tasks, while the floating point benchmarks were boosted by 21-52% and 25-53% on 4 and 8 PUs, respectively, over basic block tasks. The floating point benchmarks have more regular, loop parallelism than the integer benchmarks, as a result of which the heuristics succeed in extracting more parallelism from the floating point benchmarks.

For the integer benchmarks, the control flow heuristic improves performance 23%-54% and 23%-53% using 4 and 8 out-of-order PUs, respectively, over basic block tasks. It is important to note that the measurements shown here for the data dependence heuristic are over and above the control flow heuristic (i.e., the data dependence heuristic is applied in conjunction with the control flow heuristic). The data dependence heuristic adds modest performance improvements (<1-6% and <1-15% for 4 and 8 PUs, respectively) over the control flow heuristic.

There are many reasons for the improvements being modest: (1) Out-of-order PUs can tolerate latencies due to register communication delays significantly and (2) by including adjacent basic blocks within a task, the control flow heuristic already includes data dependence chains within tasks; the data dependence heuristic has fewer opportunities to further capture data dependences. The trends for in-order PUs are similar to those for out-of-order PUs. These improvements are better than those for out-of-order PUs because in-order PUs do not have as much latency tolerance as out-of-order PUs; the heuristics are effective in avoiding inter-task dependences, which stifle in-order PUs more than out-of-order PUs.

4.3.2. Task size

In Table 1, the columns titled Basic Block, Control Flow, and Data Dependence show the task sizes in number of dynamic instructions for the corresponding heuristic. Since only 129.compress and 145.fpppp respond to the task size heuristic, both control flow tasks and data dependence tasks are augmented with the task size heuristic

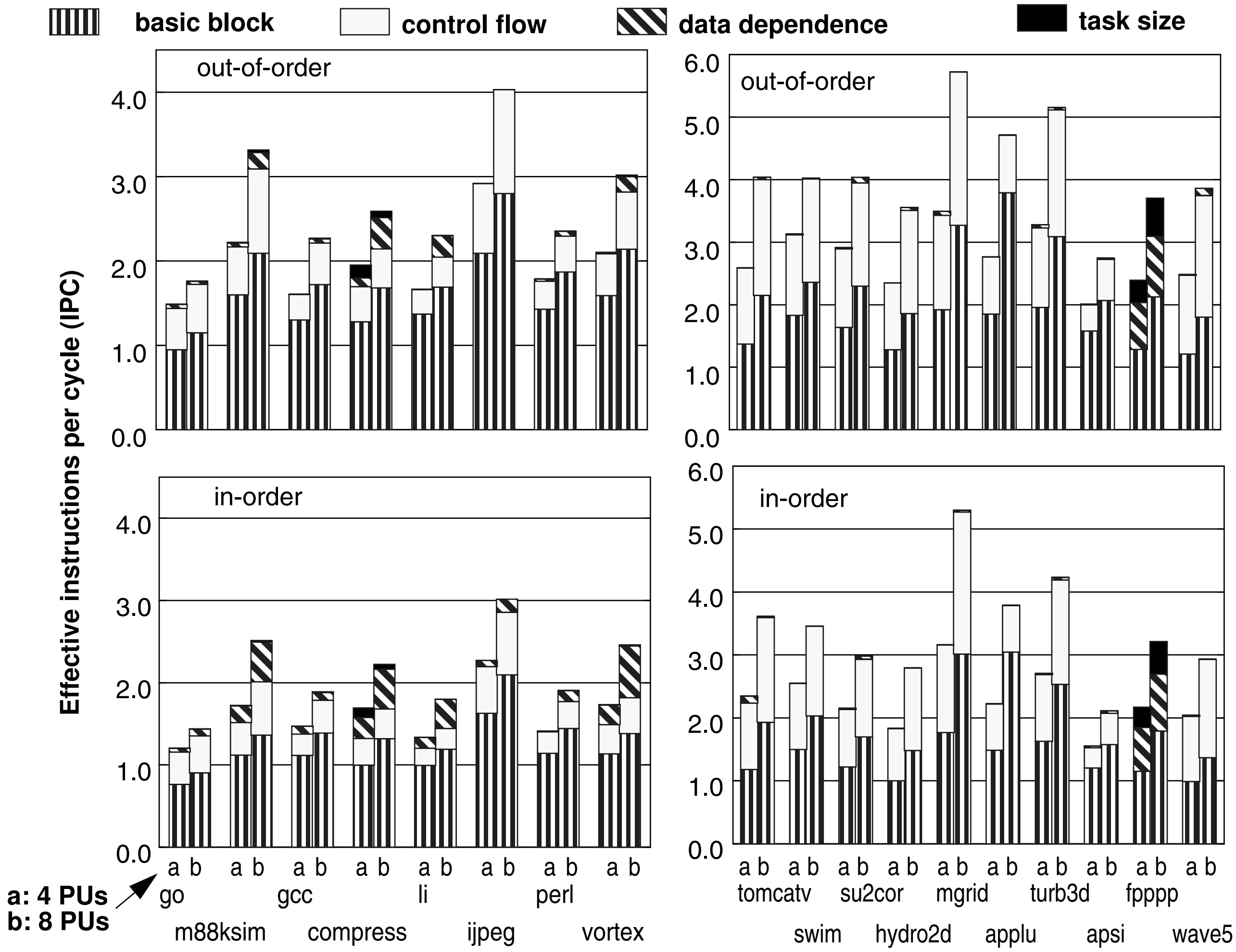

Figure 5: Impact of the compiler heuristics on SPEC95 benchmarks.

for these benchmarks. The columns titled "#dyn inst" show the number of dynamic instructions and the columns titled "#ct inst" show the number of dynamic control transfer instructions per task. We discuss the entries in the columns "task pred", "br pred", and "win span" in the next two sections. The basic block tasks contain fewer than 10 instructions for the integer benchmarks and more than 20 instructions for the floating point benchmarks (except for 104.hydro2d). In general, the control flow tasks and the data dependence tasks are larger than the basic block tasks. The data dependence tasks are smaller than other control flow tasks because the control flow heuristic greedily includes basic blocks past data dependence chains, whereas the data dependence heuristic terminates tasks as soon as a data dependence is included. 129.compress, 107.mgrid, 145.fpppp do not follow this trend because the data dependence heuristic steers task selection to paths different from the control flow heuristic, resulting in entirely different tasks. Note that due to assembly-level macro instructions, some basic block tasks may include control transfer instructions which are hidden from the compiler. For this reason, there may seem to be a discrepancy between the ratio of the size of the basic block tasks and that of the heuristic tasks and the number of control transfer instructions in the heuristic tasks.

4.3.3. Control flow speculation accuracy

Since control flow tasks and data dependence tasks usually contain multiple branches per task, comparing prediction accuracies of these tasks with those of basic block tasks requires normalizing the accuracies with respect to the average number of dynamic branches per task. The columns titled "task pred" show the task misprediction percentages and the columns titled "br pred" show the effective misprediction percentage normalized to the average number of branches per task. The prediction accuracy of the basic block tasks is higher than that of superscalar branch prediction accuracy because it includes branches, jumps, function

Benchmarks	Basic Block Tasks			Control Flow Tasks				Data Dependence Tasks				
	#dyn inst	task pred	win span	#ct inst	#dyn inst	task pred	br pred	#ct inst	#dyn inst	task pred	br pred	win span
go	6.4	14	22	2.5	18.2	15	5.8	2.0	12.7	15	7.2	53
m88ksim	4.3	3.1	25	3.0	14.8	4.0	1.4	2.4	10.3	4.9	2.0	67
gcc	5.8	4.4	39	2.5	12.4	5.8	2.3	2.3	11.6	7.4	3.2	68
compress*	5.7	5.0	38	1.8	10.2	5.7	3.2	2.8	15.0	7.8	2.8	86
li	3.9	3.3	23	1.9	8.1	4.0	2.1	1.6	7.1	5.2	3.2	43
ijpeg	10.6	6.0	63	2.4	23.3	3.7	1.5	2.4	23.8	5.1	2.1	146
perl	6.5	2.1	144	2.3	14.9	3.9	1.7	2.2	10.6	4.1	1.9	89
vortex	6.9	0.8	47	2.4	17.2	0.7	0.3	2.2	14.0	0.7	0.3	104
tomcatv	44.1	1.6	240	5.0	115	0.4	0.1	3.2	84.8	0.4	0.1	681
swim	42.0	0.1	244	4.1	87.7	0.2	0.0	4.1	87.7	0.2	0.0	703
su2cor	49.8	3.4	249	8.0	108	0.5	0.1	8.0	108	0.5	0.1	997
hydro2d	11.9	0.1	96	6.0	44.0	0.3	0.1	5.2	39.5	0.2	0.0	306
mgrid	51.4	1.1	394	2.0	106	2.2	1.1	2.0	107	2.2	1.1	665
applu	21.7	3.9	152	1.7	39.0	3.9	2.3	1.7	38.5	4.2	2.5	252
turb3d	21.2	3.4	151	2.5	41.7	5.8	2.4	2.4	40.8	6.7	2.7	246
apsi	24.8	2.9	178	2.8	51.0	4.3	1.5	2.6	46.8	4.1	1.6	311
fpppp*	958	5.6	6319	1.5	59.0	1.8	1.2	2.5	66.5	2.8	1.1	462
wave5	24.4	0.8	196	4.2	59.1	0.8	0.2	4.1	56.1	1.1	0.3	423

Table 1: Dynamic task size, control flow misspeculation rate and window span.

calls and returns. In general, the prediction hardware is able to maintain high task prediction accuracies for the control flow tasks and the data dependence tasks despite predicting one of four targets, whereas basic block tasks expose only two targets.

Comparing the basic block tasks with the control flow tasks in terms of task prediction accuracies (column "task pred"), there are two kinds of behavior: Task prediction accuracies are higher for the control flow tasks than the basic block tasks for those benchmarks which capture loop-level tasks, namely, 132.ijpeg, 101.tomcatv, 102.swim, 103.su2cor, 104.hydro2d, 107.mgrid, and 146.wave5. In these benchmarks, the most frequent tasks are loop bodies which do not expose any of the branches internal to the loop bodies and are easy to predict. The data dependence tasks have worse task prediction accuracies than the control flow task because including data dependence chains within tasks is preferred over reconverging control flow paths or loop bodies. If task prediction accuracy is normalized over the number of dynamic branches the effective prediction accuracies (column "br pred") are significantly better for the control flow tasks and data dependence tasks, demonstrating the synergy between the heuristics and the control flow speculation hardware.

4.3.4. Window span

The window span is the range of all dynamic tasks in flight in the entire processor. The average size of tasks, the number of PUs, and the control flow prediction accuracy of a program determine its window span. The window span of a program is computed using the following equation, where *Tasksize* is the average task size, *Pred* is the average inter-task control flow prediction accuracy, and *N* is the number of PUs: Although *Pred* may change slightly with increasing number of PUs, the overall effect on the window span is minimal.. In Table 1, the columns "win span" under the columns "Basic Block" and "Data Dependence" show the window span of the basic block tasks and the data dependence tasks for each of the benchmarks executing on 8 PUs. Due to the significantly smaller sizes and lower prediction accuracies, the window spans for the basic block tasks are con-

$$windowspan = \sum_{i = 0, N-1} Tasksize \times Pred^{i}$$

siderably smaller than those for the data dependence tasks. The window spans of most integer benchmarks are in the modest range of 45-140 instructions. The window spans of most floating point benchmarks is considerably larger (250-800) than those of their integer counterparts due to the larger size of tasks and higher prediction accuracy. These measurements indicate that the amount of parallelism that is exposed through branch prediction (which is used by most modern superscalar processors) is significantly less than that exposed by task-level speculation.

5. Conclusions

In this paper, we studied the fundamental interactions between sequential programs and the novel features of distributed processor organization and task-level speculation of a Multiscalar processor from the standpoint of performance. We identified important performance issues to include control speculation, data communication, data dependence speculation, load imbalance, and task overhead. We correlated these issues with a few key characteristics of tasks: task size, inter-task control flow, and inter-task data dependence. Task size affects load imbalance and overhead, inter-task control flow influences control speculation, and inter-task data dependence impacts data communication and data dependence speculation.

Task selection crucially affects overall performance achieved by a Multiscalar processor. The important heuristics to select tasks with favorable characteristics are: (1) Tasks should be neither small nor large; a small task may not expose enough parallelism and may incur overhead that may not be amortized over the execution of the task, where as a large task may incur memory dependence misspeculations and ARB overflows. (2) The number of successors of a task should be as many as can be tracked by the control flow speculation hardware; reconverging control flow paths can be exploited to generate tasks which include multiple basic blocks without taxing the prediction hardware. (3) Data dependences should be included within tasks to avoid communication and synchronization delays or misspeculation and roll back penalties. If a data dependence cannot be included within a task, then the dependence should be exposed such that the producer and consumer instructions involved in the dependence are scheduled favorably (i.e., the producer is executed early and the consumer is executed late in their respective tasks).

The task selection heuristics are effective in partitioning sequential programs into suitable tasks. The heuristics extract modest to high amount of parallelism from the integer benchmarks. The heuristics are uniformly more successful in exploiting loop-level parallelism in the floating point benchmarks. Increasing the number of PUs increases the improvements for the heuristic tasks, indicating that the heuristics better utilize extra hardware. The synergy between the heuristics and the prediction hardware is effective in improving the accuracy of control flow speculation. The window spans of data dependence tasks are significantly larger than those of basic block tasks due to their larger size and higher prediction accuracy.

Acknowledgments

We thank Scott Breach, Sridhar Gopal, and the anonymous referees for their comments and valuable suggestions on earlier drafts of this paper. We also thank the employees of Compucrafters India Private Limited, Chennai, India, V. Srinivasan, and Babak Falsafi for their help with an earlier draft of this paper.

This work was supported in part by MIP-9505853, ONR Grant N00014-93-1-0465, and by U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346, an anonymous donation to Purdue ECE, and a donation from Intel Corp. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U. S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedingsof the 19th ACM Symposium on Principles of Programming Languages*, pages 177–189, Austin, TX, Jan. 1983. Association for Computing Machinery.

[3] S. Breach, T. Vijaykumar, and G. Sohi. The anatomy of the register file in a multiscalar processor. In *Proceedingsof the 25th Annual International Symposium on Microarchitecture*, pages 181–190, San Jose, CA, Nov. 1994. Association for Computing Machinery.

[4] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30:478–490, 1981.

[5] M. Franklin. *The Multiscalar Architecture*. Ph.D. thesis, University of Wisconsin-Madison, Madison, WI 53706, Nov. 1993.

[6] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proceedings of the 19th Annual International Symposium on Com-*

puter Architecture, pages 58–67. Association for Computing Machinery, May 1992.

[7] M. Franklin and G. S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[8] P.-T. Hsu and E. Davidson. Highly concurrent scalar processing. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 386–395. Association for Computing Machinery, June 1986.

[9] Q. Jacobson, S. Bennett, N. Sharma, and J. E. Smith. Control flow speculation in multiscalar processors. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, pages 218–229, Feb. 1997.

[10] E. P. Markatos and T. J. LeBlanc. Load balancing vs. locality management in shared-memory multiprocessors. Technical Report URCSD-TR 399, University of Rochester, Oct. 1991.

[11] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 181–193, June 1997.

[12] K. Olukotun, B. A. Nayfeh, L. Hammond, K. W. n, and K.-Y. Chang. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, Oct. 1996.

[13] V. Sarkar and J. Hennessy. Partitioning parallel programs for macro-dataflow. In *Conference Proceeedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 192–201. Association for Computing Machinery, 1986.

[14] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425. Association for Computing Machinery, June 1995.

[15] SPEC newsletter, Aug. 1995.

[16] J. G. Steffan and T. C. Mowry. The potential for thread-level data speculation in tightly-coupled multiprocessors. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 2–13, Feb. 1998.

[17] J.-Y. Tsai and P.-C. Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, pages 35–46, Oct. 1996.

[18] T. N. Vijaykumar. *Compiling for the Multiscalar Architecture*. Ph.D. thesis, University of Wisconsin-Madison, Madison, WI 53706, Jan. 1998.

[19] E. Waingold et al. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, Sept. 1997.

[20] R. P. Wilson and M. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1995.