Task Selection for the Multiscalar Architecture

T. N. Vijaykumar
School of Electrical & Computer Engineering
1285 EE Building
Purdue University
West Lafayette, IN 47907
vijay@ecn.purdue.edu
Phone: 765 494 0592 Fax: 765 494 6440

Gurindar S. Sohi
Computer Sciences Department
1210 W. Dayton Street
University of Wisconsin-Madison
Madison, WI 53706
sohi@cs.wisc.edu
Phone: 608 262 7985 Fax: 608 262 9777

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. We also present a breakdown of execution times to show that register wait, load imbalance, control flow squash, and conventional pipeline losses are significant for almost all the SPEC95 benchmarks.

Task Selection for the Multiscalar Architecture

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. We also present a breakdown of execution times to show that register wait, load imbalance, control flow squash, and conventional pipeline losses are significant for almost all the SPEC95 benchmarks.

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[1]. The Multiscalar architecture [11] [12] [26] 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.

In a Multiscalar processor, sequential programs are partitioned into sequential (not necessarily independent) **tasks.** Tasks are assigned to PUs for execution by predicting inter-task control flow, similar to branch prediction. Multiple tasks are speculatively executed simultaneously on multiple PUs, but are retired in program order. All inter-task register dependences are honored by communication and synchronization [3],

1. The DEC Alpha 21264 already implements a two-cluster pipeline because bypassing across a single larger cluster would not fit within a cycle.

as specified by the compiler, and inter-task memory dependences are honored by speculation and validation in hardware [21]. Intra-task dependences are handled by the processing units, similar to superscalar processors.

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. 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. 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 Stanford Hydra [22], CMU STAMPede [27], and Minnesota Superthreaded architecture [28].

We give an overview of a Multiscalar processor, identify the key performance issues, and correlate these issues with the key characteristics of tasks in Section 2. In Section 3, we describe our compiler heuristics to select tasks with favorable characteristics. In Section 4, we report experimental results. We analyze the effects of the various heuristics on overall performance and present measurements of the key task characteristics. We empirically correlate each key characteristic with the related performance issue and isolate the impact of each of the performance issues on performance. 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 iden-

tify 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 for the Multiscalar architecture

Tasks, obtained by partitioning sequential programs, are assigned to processing units (PUs) for execution. Each PU executes the instructions of its task to completion. Simultaneous execution of multiple tasks on multiple PUs results in the completion of multiple instructions per cycle. The architecture ensures that the individual execution of each task as well as the aggregate execution of all tasks maintain the appearance of sequential program order. A combination of hardware and software mechanisms are used to ensure that control and data dependences are honored as per the original sequential program specification, regardless of what transpires in the actual parallel execution. Figure 1 illustrates the Multiscalar execution model.

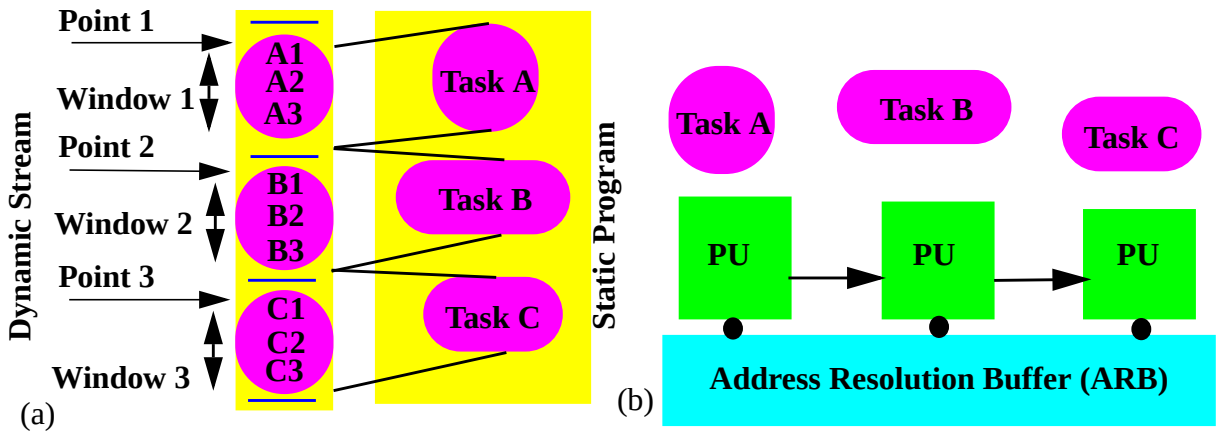

Figure 1: Abstraction of a Multiscalar processor. (a) A static program partitioned into three tasks and three points of search in the dynamic stream with three corresponding windows. (b) The tasks are assigned for execution on the processing units. The PUs are connected for communicating data and the Address Resolution Buffer detects dependence misspeculations.

2.1.1 Control flow

Dynamic prediction unravels the control flow among tasks and each predicted task is assigned to a PU for execution. Execution proceeds by assigning tasks to PUs. After assigning a task for execution, a **control flow speculation** [11] [12] [16] [23] [26] is made which predicts one of the many possible successors of the task, similar to branch prediction employed by superscalar processors [17] [25] [32] [33]. If control flow speculation is incorrect (i.e., one of the tasks assigned for execution was incorrect) then the incorrect task (and all dependent tasks[2]) is **squashed** similar to a branch misprediction in superscalar processors.

When a task is assigned for execution, the hardware needs to know the possible successors of the task; not predicting the probable successor task until its identity is known incurs performance loss. The compiler explicitly provides a list of successors of the task called **targets** and the hardware predicts one of the targets [26]. If the compiler cannot provide all of the targets of a task (e.g., indirect jumps), the hardware could learn and remember the unspecified targets as they are encountered in execution. The advantage of not specifying a target is smaller binary (and instruction memory resources) and the disadvantage is that the first time an unspecified target is encountered, the next task cannot be assigned until the target is computed, which may lead to performance loss. Apart from the successors of a task, the PU needs to know where a task ends, so that it may terminate execution. The compiler tags the instructions at the boundaries of tasks with a few extra bits called the **task-exit bits**. When the PU encounters an instruction with the task-exit bits set, it stops execution and the task ends. Branch instructions lead to two possible control flow paths and the task-exit bits identify the paths that exit the task.

2.1.2 Data values

As instructions in tasks execute, data values are produced and consumed within the same task and among different tasks, corresponding to intra-task and inter-task communication, respectively. Intra-task data values are bound to memory and register locations, similar to superscalar processors. For inter-task dependences, register dependences can be specified by simple bit-vectors since it is small, whereas the large size of memory space makes compact specification harder. It is straightforward to identify producers and consumers of register values since all register names are known statically via register specifiers. On the other hand, in the case of memory, it is difficult to determine precisely the producers and consumers of data values since memory storage names are determined dynamically via address calculations. Consequently, memory dependences are honored via **memory dependence speculation** and verification by the hardware [11] [12] [13] [14] [26] and register dependences are honored via synchronization and communication [3] [13] [26], as specified by the compiler. It is important to note that if a memory dependence is known at compile-time then the dependence need not be speculated, but may be honored via synchronization and communication, much like register dependences. Similarly, it may be advantageous to employ speculation on a register dependence, instead of synchronization and communication.

The compiler determines the set of register values that may be produced by a task and the set of register values that may be consumed by the task. If a task consumes a data value produced by another task, then

2. In a real implementation, it may be difficult to isolate dependent tasks, similar to isolating independent instructions, as mentioned before; in which case all tasks following the misspeculated task are rolled back and restarted.

the consumer task waits until the value is received. In accordance with sequential semantics, the last update of a register in a task should be **forwarded** [12] [26] to succeeding tasks[3]. If a register is guaranteed not to be modified by a task, then it may be propagated to successor tasks as and when it arrives. Since the PU cannot determine a priori which instructions comprise its assigned task (the instructions may not even have been fetched), it cannot know (1) which registers are guaranteed not to be modified, and (2) which instruction performs the last update to a register that needs to be forwarded. Waiting until all instructions in a task have executed so that all registers are updated serializes the execution of tasks, incurring performance loss. The compiler provides the set of registers which may be modified as a bit-vector called the **create mask**. The compiler determines the last update of each register and tags the instruction with a few extra bits called the **forward bits**, similar to task-exit bits. When a PU encounters an instruction with forward bits set, called a **forwarding instruction**, it forwards the value corresponding to the destination register of the instruction. If a register is forwarded in one control flow path but there are no instructions that modify the register in another path, an extra instruction called a **release instruction** may have to be inserted to send register values. The targets and the create mask of a task are collected together in its **task descriptor**.

In the case of inter-task memory data dependencies, each task speculates that it does not depend on any other task for memory values and performs 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) then the ARB detects a memory dependence violation and the offending load instruction (and dependent instructions) is **squashed**. To prevent frequent memory dependence misspeculations, a hardware mechanism to perform memory dependence prediction and synchronization, which dynamically synchronizes dependent loads and stores [21] is employed. In a real implementation, memory dependence squashes and control flow squashes may be handled identically for the purpose of simplifying hardware.

2.1.3 Sequential semantics

The original program order among the tasks is maintained by keeping track of the order in which tasks are predicted and assigned for execution on the PUs. Since the tasks are derived from a sequential program and are predicted in the original program order (similar to branch prediction), the total order among the tasks is unambiguously maintained. 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

3. If speculation and verification is employed, multiple values for a register may be sent. In that case, the value corresponding to the last update of the register should be sent last or tagged specially, to maintain sequential semantics.

state is promoted to architectural state). In a simple implementation, tasks are retired in the original program order to maintain sequential semantics; a task is retired only after its predecessor task has been retired. In a more aggressive implementation, if two tasks are completely independent, which may be guaranteed by the compiler, they can be retired out of program order. Although out of program order retiring may increase the complexity of hardware over sequential retiring, performance benefits may be accrued by relaxing the retiring order.

2.1.4 An example of a Multiscalar program

In order to clarify the kind of information required, we present an example of a code snippet and illustrate its execution. Figure 2 shows the source code and the assembly code with information specific to Multiscalar tasks. Let us assume that the loop body is partitioned into two tasks, shown by shaded regions in Figure 2(b): one from the label *Loop* to the add instruction above the label *Not_Heap* and the other from the label *Not_Heap* to the label *Continue* (the jump instruction at end of the loop body). There are many other task partitions possible but the partition chosen here serves to illustrate the kind of information required to execute tasks.

In Figure 2(c), the two branch instructions are annotated with task-exit taken bits (indicated by ST) causing the hardware to terminate the task if either of the branches are taken. The add instruction at the end also has its task-exit bits set (indicated by S) causing the task to terminate. In Figure 2(d), the jump instruction has its task-exit bit set (indicated by S) causing the task to terminate at the end of the loop.

In Figure 2(c), since all the integer instructions and the load instruction of Task1 are last updates of the respective destination registers, they have the forward bits set (indicated by F). Any register on the create mask that was not forwarded during the execution of the task is forwarded after the task terminates. For example, if the second branch of Task1 is taken then *rheap* is not forwarded by any instruction and it is automatically forwarded at the end. Task2 is similar to Task1 except for the release instruction which forwards the registers *rhit* and *rtmp*. If either of the first two branches of Task2 are taken, then the registers *rhit* and/or *rtmp* are not forwarded by any instruction; instead of letting the hardware forward them at the end (similar to *rheap* of Task1), the release instruction forwards them earlier because subsequent tasks may be waiting for them.

To explain how load and store instructions are executed, let us assume that two iterations of the loop in Figure 2(a) are executed on a four-PU configuration. Task1 and Task2 of iteration 1 and Task1 and Task2 of iteration 2 are executed on PU1, PU2, PU3, and PU4, respectively. PU1 forwards registers *ri*, *rtrace*,

```
while (i++ < n) {
        address = trace[i];
        idx = address & IDX_MASK;
        tag = address & TAG_MASK;
        if (address & HEAP_MASK)
            heap_access++;
        if (VALID(cache[idx]) &&
         (tag == TAG(cache[idx]))
            cache_hit++;
        else
            TAG(cache[idx]) = tag;
}
```

(a)

```
Loop:
        add     ri, ri, 1
        add     rtrace, rtrace, 4
        bge     ri, rn, Out
        ld      radd, 0[rtrace]
        andi    ridx, radd, IDX_MASK
        andi    rtag, radd, TAG_MASK
        andi    rtmp, radd, HEAP_MASK
        beq     rtmp, r0, Not_Heap
        add     rheap, rheap, 1

Not_Heap:
        add     rblock, rcache, ridx
        ld      rvalid, VALID[rblock]
        beq     rvalid, r0, Cache_Miss
        ld      rtmp, TAG[rblock]
        bne     rtag, tmp, Cache_Miss
        add     rhit, rhit, 1
        jmp     Continue
Cache_Miss:
        st      rtag, TAG[rblock]
Continue:
        jmp     Loop
Out:
```

(b)

Targets: Task2, Task3
Create Mask: ri, rtrace, radd, ridx, rtag, rtmp, rheap

```
Task1:
F       add     ri, ri, 1
F       add     rtrace, rtrace, 4
ST      bge     ri, rn, Task3
F       ld      radd, 0[rtrace]
F       andi    ridx, radd, IDX_MASK
F       andi    rtag, radd, TAG_MASK
F       andi    rtmp, radd, HEAP_MASK
ST      beq     rtmp, r0, Not_Heap
F S     add     rheap, rheap, 1
Not_Heap:
```

(c)

Targets: Task1
Create Mask: rblock, rvalid, rtmp, rhit

```
Task2:
F       add     rblock, rcache, ridx
F       ld      rvalid, VALID[rblock]
        beq     rvalid, r0, Cache_Miss
F       ld      rtmp, TAG[rblock]
        bne     rtag, tmp, Cache_Miss
F       add     rhit, rhit, 1
        jmp     Continue
Cache_Miss:
        release rhit, rtmp
        st      rtag, TAG[rblock]
Continue:
S       jmp     Task1
Task3:
```

(d)

Figure 2: An example of a Multiscalar program. (a) The source code of a loop of a simplified cache simulator. (b) The assembly code for the loop. The loop body is partitioned into two tasks. (c) and (d) The two tasks and their descriptors. **F**, **ST**, and **S** denote forward register, exit if taken, and exit always, respectively.

radd, *ridx*, *rtag*, and *rheap* to PU2 when the corresponding forwarding instructions are executed. In addition to forwarding registers *rblock*, *rvalid*, and *rhit*, PU2 also propagates the registers received from PU1 to PU3. PU3 consumes the registers received from PU2 and forwards new values and propagates old values for the corresponding registers. It is possible that the store instruction of PU2 (Task2, iteration 1) and the load instruction of PU4 (Task2, iteration 2) access the same memory location. Depending upon the timing of execution, either the store from PU2 reaches the ARB before the load from PU4 or vice-versa. If the store is ahead of the load then the load gets the correct value of the memory location; otherwise, when the store is executed, the ARB detects a memory dependence violation and squashes PU3 and PU4. Task1 and Task2 of iteration 2 are reexecuted on PU3 and PU4.

2.2 Multiscalar tasks: Definition

A Multiscalar task is defined to be a connected, single-entry subgraph of the static control flow graph (CFG) [1] of a sequential program. A task corresponds to a contiguous fragment of the dynamic instruction stream that may be entered only at the first instruction of the fragment. There are no other constraints on tasks except that they cannot comprise disconnected parts of the dynamic instruction stream. A task may comprise a basic block [1], multiple basic blocks, loop bodies, entire loops, or even entire function invocations. If a task contains a function call that expands to many dynamic instructions, the corresponding function definition is considered to be a part of the task. Note that more than one task may contain a call to the same function, in which case the tasks share the static code of the corresponding function definition. Arbitrary control flow and data dependences may exist among instructions of a task or different tasks; specifically, tasks are not necessarily independent. The nonrestrictive nature of tasks allows the Multiscalar architecture to exploit any grain of parallelism, ranging from instructions within a basic block to instructions of different function invocations, present in application programs.

Although tasks are defined to be static objects, there is an important relationship between the sequence of dynamic instructions corresponding to a task executed by a PU and the static task. The PU follows a particular dynamic control flow path through the static task, depending on the data values involved in the computation performed by the task. Since the compiler does not have access to dynamic control flow paths, it treats a set of static control flow paths, some of which give rise to dynamic control flow paths during execution, connected together in a subgraph of the CFG as a task. Thus, a static task, as we have defined it, is inexact in that it 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 various analyses and optimizations.

Two simple examples of tasks are: (1) a task that contains the entire program, and (2) a task that contains just one basic block. With such a wide range of options to choose from, task selection uses heuristics to steer towards tasks that deliver high performance.

2.3 Time line of a task: where is execution time spent?

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 3(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 3 (b) illustrates the various phases of scenario 2.

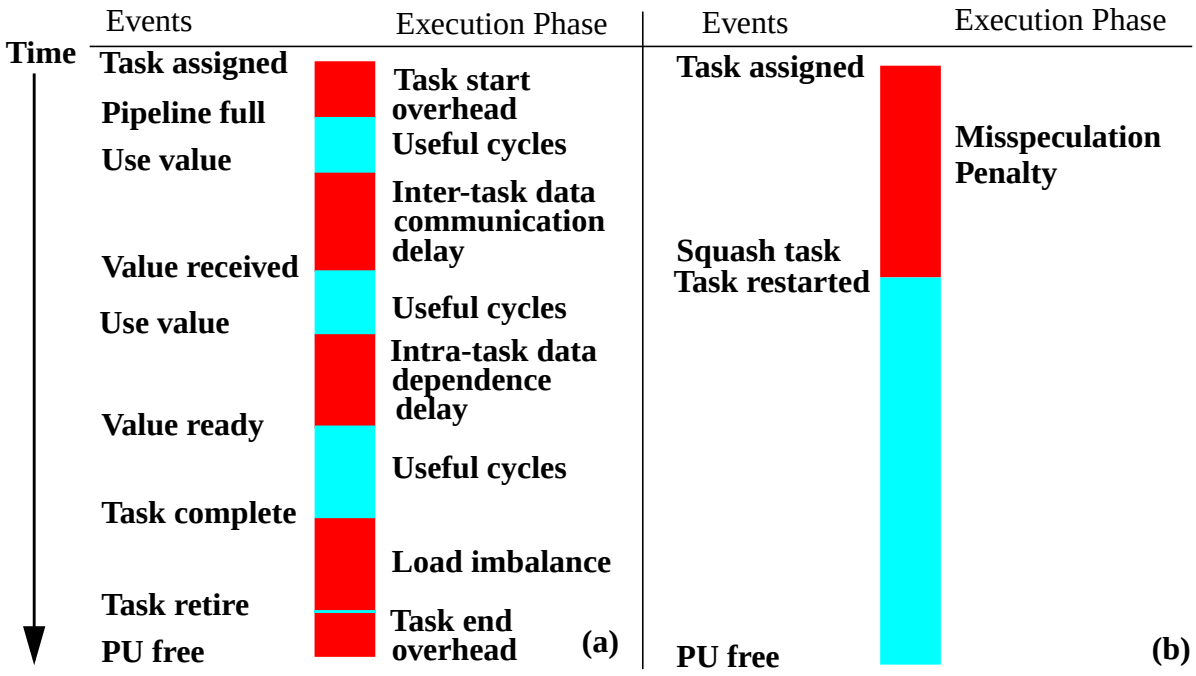

Figure 3: Time line of the execution of a task. Time extends downwards. Events are on the left and the corresponding phases are on the right of the time line. (a) Scenario 1: Task executes to completion. (b) Scenario 2: Task is squashed and restarted.

2.4 Relationship between performance issues and task characteristics

We now relate the performance issues of control flow misspeculation, inter-task data communication, memory dependence misspeculation, load imbalance, and task overhead to concrete task characteristics: task size, inter-task control flow, and inter-task data dependence.

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) 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. Large variations in the amount of computation of adjacent tasks causes load imbalance (similar to large-scale, parallel and distributed machines [5] [8] [20]) 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, extra cycles may be spent on committing the speculative state of the task to the architectural storage. The number of extra cycles may be decreased if the data is just tagged as committed as opposed to physically moving the data to a separate architectural storage.

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.

2.4.3 Inter-task data dependence

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.

3 Selecting tasks with favorable characteristics

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 [24]. 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 through the task size heuristic. 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. Basic block execution frequencies obtained by profiling were used to prioritize data dependences for the data dependence heuristic. The profiler also provided dynamic instruction count per function to the task size heuristic to include entire function calls within tasks.

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 4 shows the heuristics in pseudo-code. The function 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

```
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));
}
is_a_terminal_edge(blk, ch) {
  return(dfs_num(blk) < dfs_num(ch));
}
```

```
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 < TARGET_NUM) {
              feasible_task(root, blk, ch);
      } else {
          add_to_task_q(ch);
      }
  } else {
      for each child ch of blk {
          add_to_task_q(ch);
      }
  }
}
```

Figure 4: Task selection heuristics. task_selection() is the top level driver. task_size_heuristic() unrolls loops and marks function calls to be included within tasks. identify_data_dependences() determines all the register dependences and simple, named memory dependences. dependence_task() explores one basic block per invocation and queues the children of the basic block under consideration for further exploration. dependence_task() invokes is_a_terminal_node() and is_a_terminal_edge() which determine whether a node and an are terminal, respectively. feasible_task() tracks the basic blocks that correspond to fewer than TARGET_NUM successors. codependent() determines whether a basic block is in the codependence set of a data dependence edge or not.

blocks are assembled into one task. 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 com-

putation 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.

The function task_size_heuristics() in Figure 4 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) instructions[4] are unrolled to expand to LOOP_THRESH instructions. Note that these thresholds do not limit the size of all tasks, but only in certain cases. These threshold values were found to avoid the problems of both small and large tasks after experimentation with the SPEC95 benchmarks.

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. The functions is_a_terminal_node() and is_a_terminal_edge() in Figure 4 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.)

Apart from the number of successors, predictability of successors is an important factor in control flow speculation. By marking loop back edges as terminal, the control flow heuristic favors exposing the (usually) predictable loop back edges to the prediction hardware. Profiling may also be used to identify the branches that are difficult to predict so that the control flow edges resulting from such branches are included within tasks. Since our profiling infrastructure does not track prediction accuracy on a per-branch basis, the current implementation of the control flow heuristic does not use branch prediction profile information.

4. The current implementation uses static instruction count, but static profiling can be used to incorporate dynamic instruction count.

The function dependence_task() in Figure 4 shows how the number of successors of a task is controlled. 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. 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 [10], predication [15] [19], and if-conversion [2] [30]. 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 effectiveness of the heuristics by getting rid of branches 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. Further discussion of scheduling register communication is beyond the scope of this paper.

3.4 Data dependence heuristic

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

Register dependences are identified and specified entirely by the compiler using traditional def-use dataflow equations [1]. The main impediment to including a memory dependence within a task is that many memory dependences are unknown or ambiguous at compile-time. The compiler identifies simple, unambiguous memory dependences involving named, non-pointer accesses. For the rest of the (ambiguous) dependences, the compiler speculates that there are no dependences relying on the hardware (the ARB and memory dependence synchronization mechanism [21]) for correctness and performance.

Instead of our approach of including basic blocks within tasks, another possible approach is to use global code motion to move the instructions that are not included in a task but that are dependent on the task. Such code motion can not be speculative and can be done only if the compiler can guarantee that no dependences (register or memory) are violated. There are many compiler pointer analysis techniques which work well for programs that do not employ intricate heap pointers [6] [4] [18] [7] [9] [31]. Due to the prevalence of heap structures in most of our benchmarks, memory dependence ambiguities pose a severe restriction on the ability of the compiler to perform such global code motion.

The function dependence_task() in Figure 4 integrates the data dependence heuristic with the control flow heuristic. For each data def-use dependence, starting from the highest priority (most frequent) 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. 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. More details on the heuristics are in [29].

3.5 Description of tasks selected by the heuristics

Figure 5 illustrates task partitions that may result when a data dependence edge is considered.

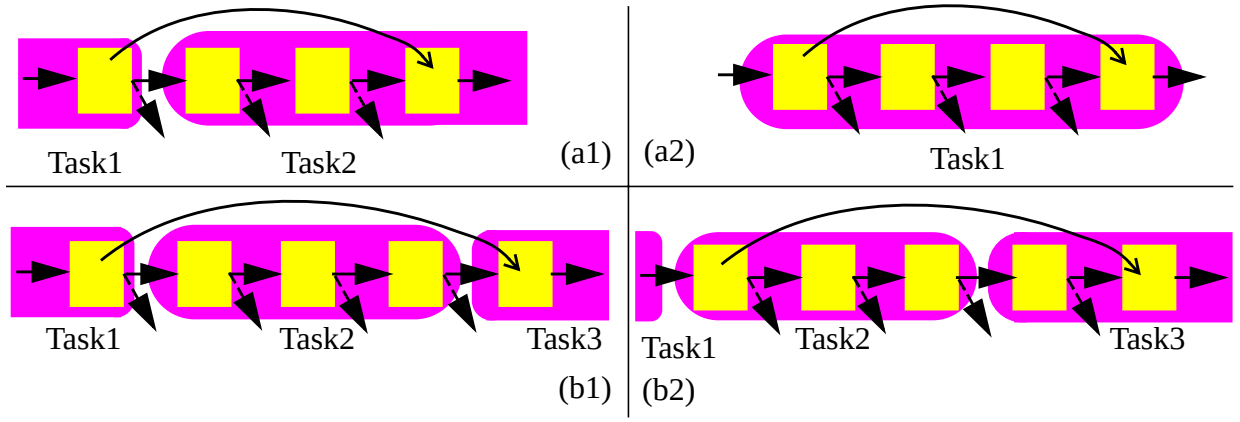

Figure 5: Task selection and data dependence edges. (a1) A part of the CFG of a program. ➔ indicates a control flow edge and ⟶ indicates a data dependence edge. Dashed arrows represent terminal control flow edges and shaded regions represent tasks. (a2) A task selection resulting in one task that includes the data dependence edge and has five successors. (b1) A task selection resulting in three tasks with poorly scheduled inter-task data dependence. (b2) A task selection resulting in three tasks with favorably scheduled inter-task data dependence.

Figure 5(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. For this example, let us assume that the maximum number of successors is four. Since the control flow heuristic does not take data dependences into consideration, the dependence is split between Task1 and Task2. Figure 5(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 5(b1) shows a task partition obtained by the control flow heuristic, assuming the number of hardware targets to be four. 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 5(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

An important goal of this paper is to evaluate performance of the combination of a Multiscalar processor and the compiler on large benchmarks. To that end, the heuristics described in the preceding sections have been implemented in the Gnu C Compiler, gcc. SPEC95 benchmark 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 and produces output result files for verification. The simulator executes all instructions of the benchmarks except for system calls, in order to maintain high accuracy of performance results.

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. To get a better understanding of the importance of the various performance factors, we report a breakdown of execution time corresponding to the following categories: useful computation, register wait, load imbalance, memory synchronization wait, memory dependence squash, control flow squash, overhead and conventional (intratask) pipeline losses due to branch mispredictions and memory hierarchy. 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 each of 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, the memory hierarchy consisting of the ARB, L1 data and instruction cache, L2 cache and main memory. Apart from these hardware components the simulator models the interconnection between the processing units and the ARB and L1 instruction and data caches, the L1 caches and the L2 caches, 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 interconnects. Hardware parameter values used by us are summarized in Table 1.

Table 1: Hardware parameter values used in our experimental evaluation.

Component	Description
PUs	2-way issue, 16-entry reorder buffer, 8-entry issue list
FUs	2 integer, 1 complex, 1 floating point, 1 branch, 1 memory
Intra-task prediction	gshare with 16-bit history, 64K-entry table of 2-bit counters
Inter-task prediction	path-based with 16-bit history, 64K-entry table of 2-bit counters and 2-bit target numbers
Register Ring	2 values per cycle, bypass same cycle between adjacent PUs
L1 I-cache	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, augmented with a 32KB, 2-way associative task cache
L1 D-cache	64KB (4PU)/128KB (8PU), 2-way associative, 32 byte blocks, 2 cycle hit, interleaved as many banks as the number of PUs, lock-up free, fully pipelined
ARB	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, augmented with a 256-entry memory synchronization table
L2 cache	4MB, 2-way associative, 12 cycle hit, 16 bytes per cycle transfer
Main memory	Infinite capacity, 58 cycle latency, 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 [29]. Register communication scheduling is not discussed in this paper; details are available in [29].

We use the SPEC95 benchmark suite throughout our experimental evaluation. Profiling was done using the inputs specified by the SPEC95 suite. The profiler provided basic block execution frequencies to allow the data dependence heuristic to prioritize data dependences and dynamic instruction count per function to the task size heuristic to include entire function calls within tasks. Note that in some cases the SPEC standard test inputs are used to profile and the SPEC standard train inputs are used to run, to keep the simulation runs to fewer than many billion instructions. All the results were obtained by running the programs to completion.

4.3 Experiments and Results

We report experimental results on overall performance followed by measurements of task characteristics.

4.3.1 Effectiveness of the compiler heuristics on overall performance

Figure 6 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 combined 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 19-64% and 32-57% on 4 and 8 PUs, respectively, while the floating point benchmarks were boosted by 21-101% and 25-116% 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 19-54% and 23-43% 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

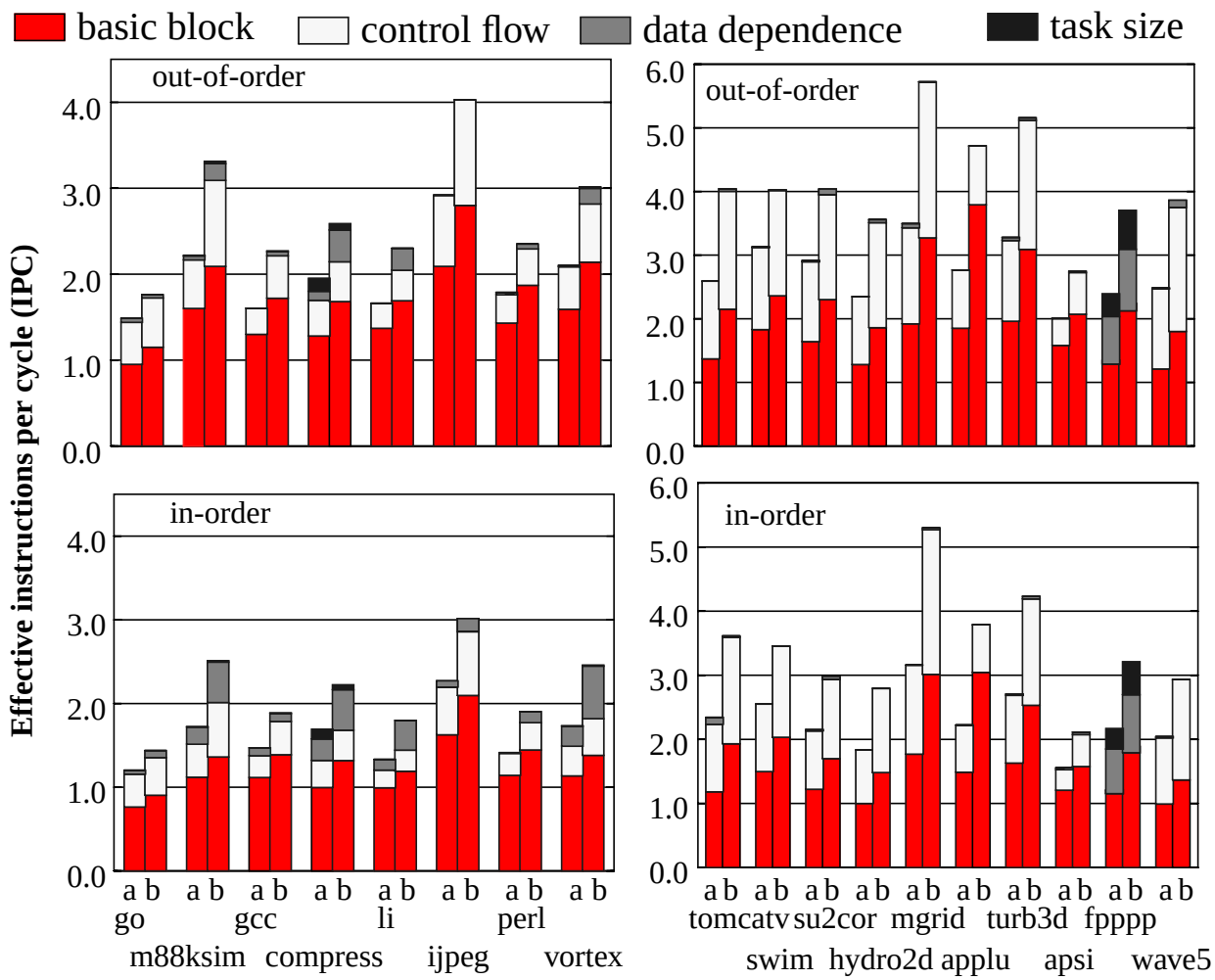

Figure 6: Impact of the compiler heuristics on SPEC95 benchmarks. The graphs are annotated to show use of out-of-order or in-order PUs. The two experiments marked a and b use 4 and 8 PUs, respectively.

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 2, the columns titled Basic Block, Control Flow, and Data Dependence show the task sizes in number of dynamic instructions for the corresponding heuristic. 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

Table 2: Dynamic task size, control flow misspeculation rate and window span for SPEC95 benchmarks. Since only 129.compress and 145.fpppp respond to the task size heuristic, both control flow tasks and data dependence tasks are augmented with task size heuristic for these benchmarks.

Bench marks	Basic Block			Control Flow					Data Dependence				
	#dyn inst	task pred	win span	#ct inst	#dyn inst	task pred	br pred	win span	#ct inst	#dyn inst	task pred	br pred	win span
go	6.4	14	32	2.5	18.2	15	5.8	88	2.0	12.7	15	7.2	62
m88ksim	4.3	3.1	31	3.0	14.8	4.0	1.4	103	2.4	10.3	4.9	2.0	70
gcc	5.8	4.4	40	2.5	12.4	5.8	2.3	81	2.3	11.6	7.4	3.2	72
compress*	5.7	5.0	38	1.8	10.2	5.7	3.2	67	2.8	15.0	7.8	2.8	92
li	3.9	3.3	28	1.9	8.1	4.0	2.1	56	1.6	7.1	5.2	3.2	47
ijpeg	10.6	6.0	69	2.4	23.3	3.7	1.5	164	2.4	23.8	5.1	2.1	160
perl	6.5	2.1	48	2.3	14.9	3.9	1.7	104	2.2	10.6	4.1	1.9	74
vortex	6.9	0.8	54	2.4	17.2	0.7	0.3	134	2.2	14.0	0.7	0.3	109
tomcatv	44.1	1.6	333	5.0	115	0.4	0.1	907	3.2	84.8	0.4	0.1	669
swim	42.0	0.1	335	4.1	87.7	0.2	0.0	697	4.1	87.7	0.2	0.0	697
su2cor	49.8	3.4	354	8.0	108	0.5	0.1	849	8.0	108	0.5	0.1	849
hydro2d	11.9	0.1	95	6.0	44.0	0.3	0.1	348	5.2	39.5	0.2	0.0	314
mgrid	51.4	1.1	396	2.0	106	2.2	1.1	785	2.0	107	2.2	1.1	793
applu	21.7	3.9	152	1.7	39.0	3.9	2.3	273	1.7	38.5	4.2	2.5	266
turb3d	21.2	3.4	151	2.5	41.7	5.8	2.4	273	2.4	40.8	6.7	2.7	259
apsi	24.8	2.9	179	2.8	51.0	4.3	1.5	352	2.6	46.8	4.1	1.6	325
fpppp*	958	5.6	6318	1.5	59.0	1.8	1.2	443	2.5	66.5	2.8	1.1	483
wave5	24.4	0.8	189	4.2	59.1	0.8	0.2	460	4.1	56.1	1.1	0.3	432

dependence tasks are smaller than the 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 Inter-task 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 superscalar branch prediction accuracy because it includes branches, jumps, function 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 2, the columns “win span” under the columns “Basic Block” and “Data Dependence”

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

show the window span of the basic block tasks, control flow tasks, and 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 considerably smaller than those for the data dependence tasks. The window spans of most integer benchmarks are in the modest range of 47-160 instructions. The window spans of most floating point benchmarks is considerably larger (259-849) than those of their inte-

ger 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.

4.3.5 Breakdown of execution time into performance components

In Figure 7, we present the breakdown of execution time in terms of key performance issues. The categories are: useful computation, register wait, load imbalance, memory synchronization wait, memory dependence squash, control flow squash, task overhead, and the rest of the CPI. The rest of the CPI includes intra-task branch misprediction, memory hierarchy components and other conventional pipeline losses. Although the contribution of each factor varies across benchmarks, register wait, load imbalance, control flow squash, and conventional pipeline losses are significant for almost all the benchmarks.

Since out-of-order PUs can overlap delays, these measurements were obtained by monitoring the commit point of the PUs. For example, if the instruction at the commit point waits for register values and no other useful work is done in that cycle, then the cycle is charged to register communication delay. This method gives only the apparent number of cycles spent and not the exact cycle count accountable to each performance issue separately because there could be other instructions whose delay is overlapped with the instruction at the commit point[5]. Thus, although we measure the overall CPI accurately, our method does not isolate the individual components exactly[6]. To avoid any confusion, we label the breakdown of overall CPI into components as “observed cycles per instruction” in Figure 7.

Register wait times for 4 and 8 PUs are similar for most benchmarks because most register values are short-lived and are required only in the next successor task and few are used by tasks farther in the future. Since the overall CPI decreases on increasing the number of PUs, inter-task register dependences become more important performance factors for larger number of PUs. For the control flow tasks, register wait time is 10-24% and 17-40% of total execution time using 4 and 8 PUs, respectively. For the data dependence tasks, register wait time is 0-24% and 12-40% of the total execution time using 4 and 8 PUs, respectively. The data dependence tasks exhibit unusually large reduction in register wait time for 145.fpppp because the task size heuristic splits the enormous basic blocks of 145.fpppp into smaller blocks, which results in dense register dependences. Overall, the data dependence tasks incur fewer wait cycles indicating that by

5. For example, if register communication delay is measured to be 20% of overall execution time, performance may not improve by 12.5% if register communication is completely removed because there may be other instructions waiting due to memory dependences overlapped with the instructions waiting for register values.
6. To get the exact cycle counts, each of the performance factor would have to be simulated as perfect separately, which increases the number of simulation runs inordinately.

including data dependence chains within tasks, the heuristics are able to reduce register communication delay.

Memory dependences usually span larger number of dynamic instructions than the window span established by basic block tasks. Consequently, basic block tasks do not incur any significant memory synchronization wait. Although the implementation of the data dependence heuristic takes into account only a few memory dependences and all register dependences, there is no significant difference in memory synchronization wait times between the control flow tasks and the data dependence tasks. The integer benchmarks incur few memory dependence squashes due to the small sized tasks. For the floating point benchmarks, hardware memory dependence synchronization is effective in keeping memory dependence squashes in check. In 124.m88ksim and 145.fpppp a few memory dependences, which are not identified by simple compiler analyses, are exposed in the data dependence tasks causing an increase in memory synchronization wait time. The control flow tasks avoid this problem, indicated by the significantly less memory synchronization wait time as measured by our simulator.

In general, the control flow tasks and the data dependence tasks exhibit less load imbalance than the basic block tasks due to the amortization of load imbalance loss over larger tasks. For the integer benchmarks, the control flow heuristic introduces disparity by creating large tasks with non-terminal edges and small tasks with terminal edges where as the data dependence heuristic terminates tasks as soon as a dependence chain is included, naturally distributing the load evenly. For the floating point benchmarks, load imbalance for the control flow tasks and the data dependence tasks is caused due to variation in the amount of computation done in loop iterations. Load imbalance using 4 and 8 PUs are similar for most benchmarks because adjacent dynamic tasks contribute to overall load imbalance more than variation in dynamic tasks that are farther apart. Since the overall CPI decreases on increasing the number of PUs, load imbalance becomes a more important performance factor, as a percentage of overall CPI, for larger number of PUs. For the control flow tasks, load imbalance accounts for 9-16% and 12-25% of total execution time using 4 and 8 PUs, respectively. For the data dependence tasks, load imbalance accounts for 9-16% and 12-24% of total execution time using 4 and 8 PUs, respectively.

Form the point of view of control flow, there is no significant difference between the control flow tasks and the data dependence tasks because both control the number of successors of tasks. For the integer benchmarks, control flow squashes account for 0-18% and 0-28% of total execution time using 4 and 8 PUs, respectively. There is a strong correlation between speculation accuracy and wasted cycles. Control flow

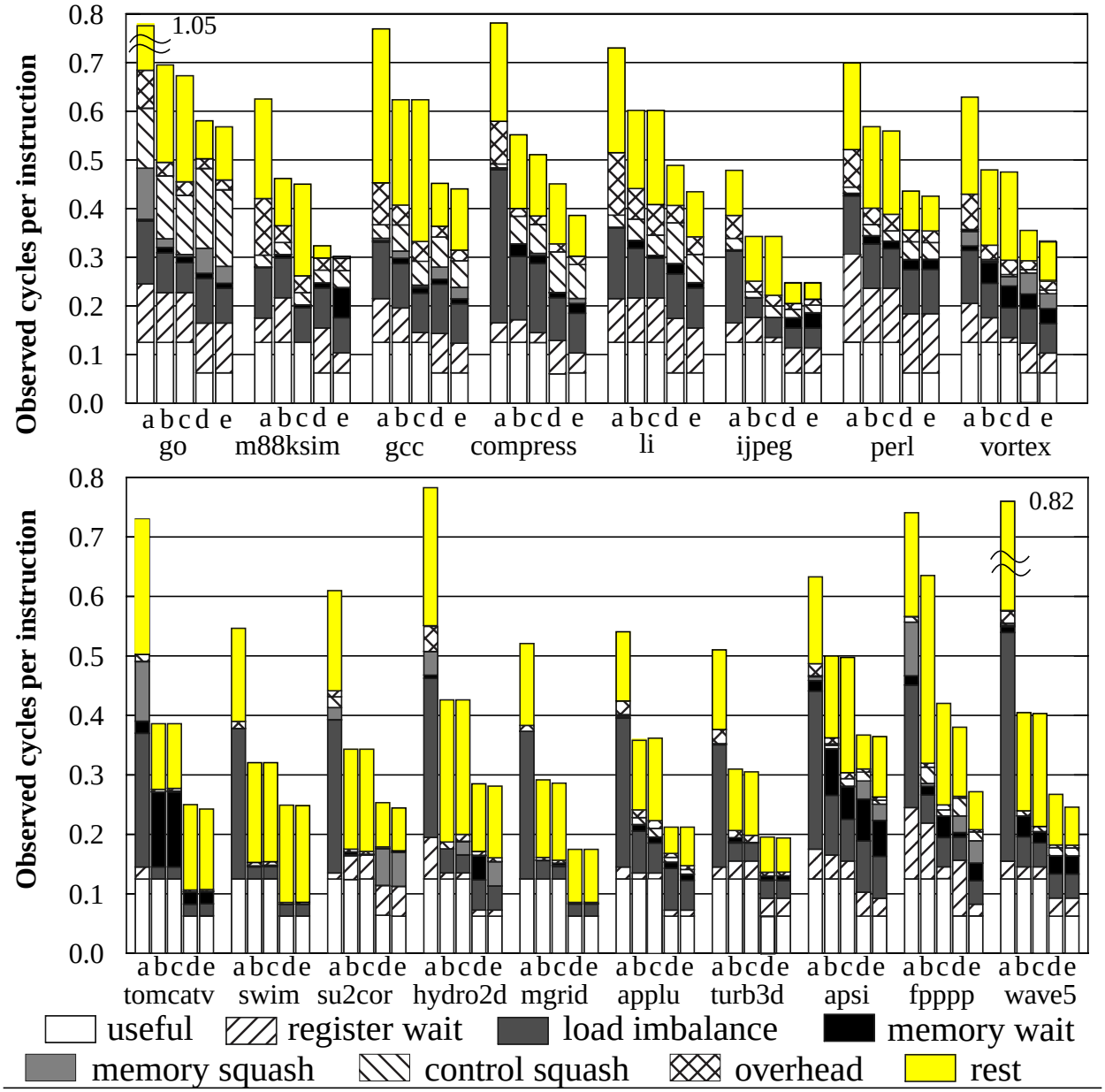

Figure 7: Breakdown of execution time in terms of key performance issues. The five experiments marked a, b, c, d, and e use basic block tasks, control flow tasks on 4 PUs, data dependence tasks on 4 PUs, control flow tasks on 8 PUs, and data dependence tasks on 8 PUs, respectively. For 129.compress and 145.fpppp, control flow and data dependence tasks are augmented with the task size heuristic.

speculation accuracy is high for 147.vortex and all the floating point benchmarks and correspondingly these benchmarks incur few wasted cycles due to control squashes.

Task overhead decreases significantly for the control flow tasks and the data dependence tasks over basic block tasks due to their larger size, for the integer benchmarks. Since even the basic block tasks are large

for the floating point benchmarks, the programs incur little overhead. Increasing the number of PUs decreases task overhead because of overlap of overhead time among more PUs.

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 determined and explored the key implications of the interactions for the compiler. 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. Although the contribution of each factor varies across benchmarks, register wait, load imbalance, control flow squash, and conventional pipeline losses are significant for almost all the benchmarks. Inter-task register communication delay and load imbalance are more important performance factor for larger number of PUs. Task overheads decrease significantly due to the larger sizes of the heuristic tasks. Task overheads are less significant as the number of PUs increases. The synergy between the heu-

ristics 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.

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 *Proceedings of 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 *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 181–190, San Jose, CA, Nov. 1994. Association for Computing Machinery.

[4] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–310. Association for Computing Machinery, June 1990.

[5] T. C. K. Chou and J. A. Abraham. Load balancing in distributed systems. *IEEE Transactions on Software Engineering*, SE-8(4):401–412, July 1982.

[6] D. S. Coutant. Retargetable high-level alias analysis. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 110–118. Association for Computing Machinery, 1986.

[7] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 230–239. Association for Computing Machinery, 1994.

[8] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.

[9] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, 1994.

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

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

[12] 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 Computer Architecture*, pages 58–67. Association for Computing Machinery, May 1992.

[13] M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 236–245, Portland, OR, Dec. 1992. Association for Computing Machinery.

[14] 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.

[15] 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.

[16] 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.

[17] Q. Jacobson, E. Rotenberg, and J. Smith. Path-based next trace prediction. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 14–23. Association for Computing Machinery, Dec. 1997.

[18] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer analysis. In *Proceedings of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–248. Association for Computing Machinery, June 1992.

[19] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, Portland, OR, Dec. 1992. Association for Computing Machinery.

[20] E. P. Markatos and T. L. Blanc. Load balancing vs. locality management in shared-memory multiprocessor. In *Proceedings of the 1992 International Conference on Parallel Processing*, volume I, Architecture, pages I:258–267, Boca Raton, Florida, Aug. 1992. CRC Press.

[21] 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.

[22] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, 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.

[23] D. Pnevmatikatos, M. Franklin, and G. S. Sohi. Control flow prediction for dynamic ILP processors. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 153–163, Austin, TX, Dec. 1993. Association for Computing Machinery.

[24] 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.

[25] J. E. Smith. A study of branch prediction strategies. In *Proc. of the 8th Annual Symposium on Computer Architecture*, pages 135–148, May 1981.

[26] 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.

[27] 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.

[28] 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.

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

[30] N. Warter, S. Mahlke, W. Hwu, and B. Rau. Reverse if-conversion. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–299. Association for Computing Machinery, June 1993.

[31] 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.

[32] T.-Y. Yeh and Y. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51–61. Association for Computing Machinery, Nov. 1991.

[33] T.-Y. Yeh and Y. Patt. Alternative implementations of two-level adaptive training branch prediction. In *Proceedings of the 19 Annual International Symposium on Computer Architecture*. Association for Computing Machinery, May 1992.