Microarchitectural Innovations: Boosting Microprocessor Performance Beyond Semiconductor Technology Scaling

ANDREAS MOSHOVOS, MEMBER, IEEE, AND GURINDAR S. SOHI, MEMBER, IEEE

Invited Paper

Semiconductor technology scaling provides faster and more plentiful transistors to build microprocessors, and applications continue to drive the demand for more powerful microprocessors. Weaving the "raw" semiconductor material into a microprocessor that offers the performance needed by modern and future applications is the role of computer architecture. This paper overviews some of the microarchitectural techniques that empower modern high-performance microprocessors. The techniques are classified into: 1) techniques meant to increase the concurrency in instruction processing, while maintaining the appearance of sequential processing and 2) techniques that exploit program behavior. The first category includes pipelining, superscalar execution, out-of-order execution, register renaming, and techniques to overlap memory-accessing instructions. The second category includes memory hierarchies, branch predictors, trace caches, and memory-dependence predictors. The paper also discusses microarchitectural techniques likely to be used in future microprocessors, including data value speculation and instruction reuse, microarchitectures with multiple sequencers and thread-level speculation, and microarchitectural techniques for tackling the problems of power consumption and reliability.

Keywords*—*Branch prediction, high-performance microprocessors, memory dependence speculation, microarchitecture, out-of-order execution, speculative execution, thread-level speculation.

I. INTRODUCTION

Microprocessor performance has been doubling every year and a half for the past three decades, in part due to semiconductor technology scaling, and in part due to innovations in computer architecture and accompanying software. Semiconductor technology scaling has resulted in larger numbers of smaller and faster transistors. Innovations in computer architecture have resulted in microprocessors that achieve performance greater than what would have been possible by technology scaling alone.

There are two main reasons why architecture is instrumental in boosting performance beyond technology scaling. First, technology scaling is often nonuniform. For example, the technologies used to build processors are optimized for speed, while the technologies used to build main memories are mostly optimized for density. Without the help of novel architectural techniques, a shrunk and hence faster version of a processor would simply spend most of its time waiting for the relatively slower memory. Second, technology scaling facilitates higher integration by allowing us to pack more transistors on a chip of the same size. These additional transistors, in turn, facilitate architectural innovations that would not be possible otherwise. Many of these innovations are meant to increase the concurrency of instruction processing. Thus, the techniques used to address the above two issues can broadly be classified into two categories: 1) techniques to deal with the mismatch created by nonuniform technology scaling and 2) techniques to increase the concurrency in instruction processing.

In this paper, we review some of the microarchitectural techniques that are implemented in most modern high-performance processors. We also discuss recent proposals that may be implemented in future processors. We focus on providing the underlying motivation and on explaining the principles underlying their operation. Our emphasis is on *microarchitectural techniques*, i.e., techniques that are, in general, not directly visible to a programmer, and consequently require no support from external entities. These techniques improve performance without placing an additional burden to software developers.

Manuscript received January 14, 2001; revised June 15, 2001. The work of A. Moshovos was supported in part by the University of Toronto and by the National Sciences and Engineering Research Council of Canada. The work of G. S. Sohi was supported in part by the National Science Foundation under Grants CCR-9900584 and EIA-0071924, by the University of Wisconsin Graduate School, and by a donation from Intel Corp.

A. Moshovos is with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada (e-mail: moshovos@eecg.toronto.edu).

G. S. Sohi is with Computer Sciences, University of Wisconsin-Madison, Madison, WI 53706 USA (e-mail: sohi@cs.wisc.edu).

Publisher Item Identifier S 0018-9219(01)09682-7.

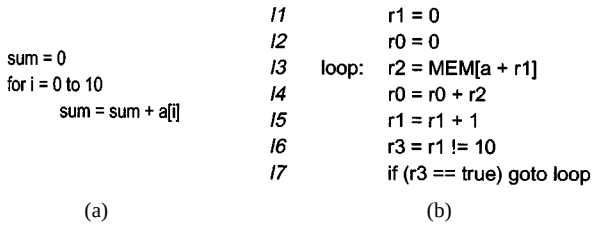


Fig. 1. (a) A simple loop calculating the sum of all ten elements of array $a[\cdot]$. (b) Machine-code level implementation of this loop. Sum and i have been register allocated to registers r0 and r1, respectively.

In Section II, we present techniques to increase the concurrency in instruction processing. In Section III, we discuss techniques that exploit regularities in program behavior. Section IV illustrates the use of the above techniques in a typical modern high-performance processor. In Section V, we discuss microarchitectural techniques that are likely to be found in future microprocessors.

II. Increasing Concurrency in Instruction Processing

Many of the microarchitectural mechanisms found in a modern microprocessor serve to increase the concurrency of instruction processing. To understand the rationale behind and progression of these techniques, let us begin with the incumbent model for specifying a program.

The most widespread model is the *sequential execution model*. Consider, for example, the loop shown in Fig. 1(a), which calculates the sum of the elements of array $a[\cdot]$. This code is written with an assumption of *sequential execution semantics* or simply *sequential semantics*. That is, we assume that statements will execute one after the other and in the order they appear in our program. For example, we expect that "$\mathrm{sum} = 0$" will execute before "$\mathrm{sum} = \mathrm{sum} + a[0]$," and that the latter will execute before "$\mathrm{sum} = \mathrm{sum} + a[1]$" and so on. Consequently, when our code is translated into machine instructions as shown in Fig. 1(b), *sequential semantics* implies that instructions have to execute sequentially, in program order: I1 executes before I2, while I3 of iteration 1 executes before I3 of iteration 2.

The sequential execution model is an artifact of how computers were implemented several decades ago. It is the incumbent model even today because it is simple to understand, reason about, and program. Moreover, it results in an unambiguous and repeatable execution: repeated executions of the same program with the same initial state results in the same state transitions. This property is critical to the engineering and debugging of both software and hardware.

Sequential instruction processing, however, is at odds with high performance. To achieve high performance we need to overlap the processing of instructions, i.e., process multiple instructions in parallel. Many of the microarchitectural techniques, therefore, serve to: 1) allow overlapped and parallel instruction processing, yet retain an external appearance of sequential instruction processing and 2) increase available concurrency so that parallel and overlapped instruction processing can be achieved.

The processing of instructions can be overlapped via pipelining, which we discuss in Section II-A. Further overlap can be achieved with techniques for *instruction-level parallelism (ILP)*, which we discuss in Section II-B. An observation critical to parallel execution of a sequential program is that the (sequential) semantics of a program define what should *appear* to take place when a program is executed and not necessarily how instruction processing need take place internally. Since the only way an external entity (e.g., a programmer) can determine whether an instruction has executed is by inspecting the machine's state, only updates to the machine state, and not the actual processing of instructions, need to adhere to sequential execution semantics.

The amount of parallel execution that can be achieved is limited by the concurrency amongst the instructions available for processing. The concurrency is limited by constraints that impose an ordering on instruction processing. Many microarchitectural techniques serve to increase concurrency by relaxing ordering constraints. (We mention these constraints when we discuss the techniques used to relax them.)

A. Pipelining

Fig. 2(a) shows how instruction processing would proceed sequentially. As shown in Fig. 2(b), processing a single instruction involves a number of micro-operations. *Pipelining* overlaps the micro-operations required to execute different instructions. Instruction processing proceeds in a number of steps, or *pipeline stages* where the various micro-operations are executed. In the example of Fig. 2(c), instruction processing proceeds into three steps: fetch, decode, and execution and the processing of I3, I4, and I5 is now partially overlapped.

Even though instruction processing is overlapped, an external appearance of sequential execution is maintained. Specifically, at any given point in time the machine state is such that it appears that *all* instructions up to a specific point have been executed, while *none* of the instructions that follow have been executed. For example, at point a in Fig. 2(c), I4 has completed execution while I5 has been decoded and I6 has been fetched. As far as an outside observer is concerned, I5 and I6 have not been processed since they have not updated any registers, memory or any other visible state.

With pipelining an instruction can be in two states: *in-progress* and *committed*. In-progress instructions may have been fetched, decoded or performed some actions, however, they have not made any changes to user-visible machine state. *Committed* instructions have completed execution and made changes to machine state. As far as a user is concerned only committed instructions have executed; in-progress instructions appear as if they have not been processed at all.

Pipelining improves instruction processing throughput, but does not reduce the latency of instruction processing. In general, none of the techniques that attempt to overlap instruction processing reduce the latency of processing an individual instruction; typically they sacrifice instruction processing latency for instruction processing throughput.

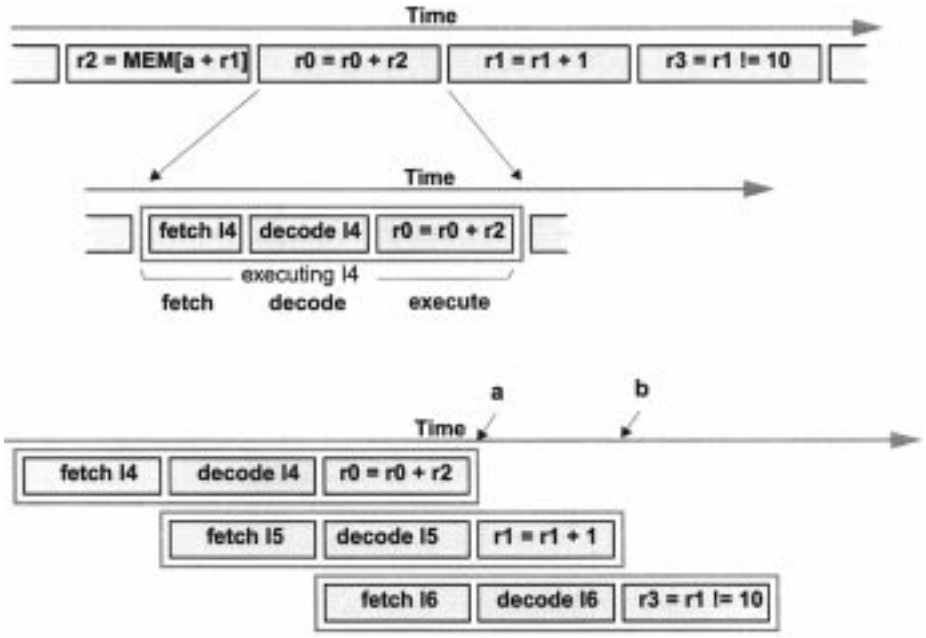


Fig. 2. (a) Execution without pipelining. (b) Steps required for executing a single instruction. (c) Execution with pipelining: the various execution steps of different instructions are overlapped.

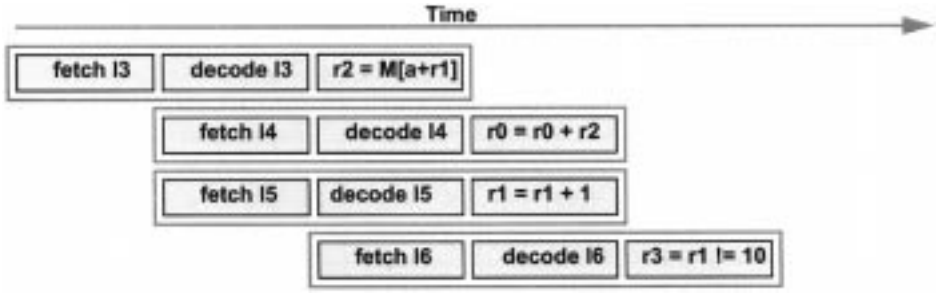


Fig. 3. Superscalar execution. I4 and I5 execute in parallel. I4 has to wait for I3 (since it requires the value in r2). I6 has to wait for I4 as it needs the new value of r1 produced by I4.

Additional information about pipelining can be found in [19] and [29].

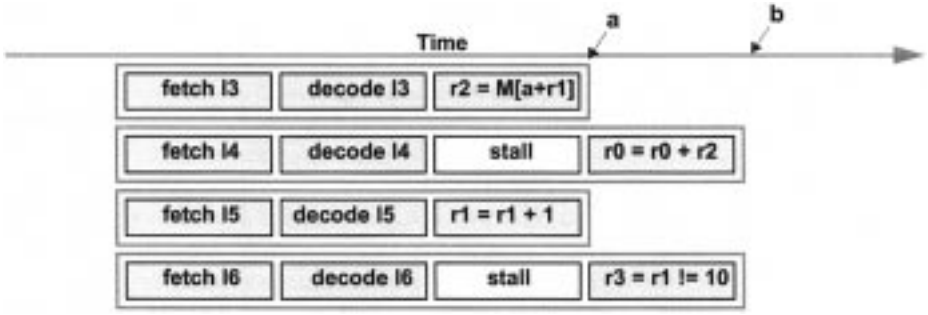
B. Instruction Level Parallelism

The simple pipelining approach achieves some overlap in instruction processing, but allows only one instruction in any phase of processing (e.g., decode). Moreover, instructions remain in their original program order as they flow through the pipeline stages: instructions are decoded, executed, and write their results in program order. The techniques we describe next attempt to further increase the concurrency of instruction processing. Superscalar execution allows multiple instructions, that are adjacent in program order, to be in the same stage of processing simultaneously. Out-of-order execution allows instructions to bypass other instructions, and enter various stages of instruction processing (e.g., execution) in an order that is different from the original program order. Speculative execution increases the available concurrency by alleviating ordering constraints (e.g., control dependences). The key in these approaches is to process instructions in an arbitrary order, but give the external appearance that they were processed sequentially.

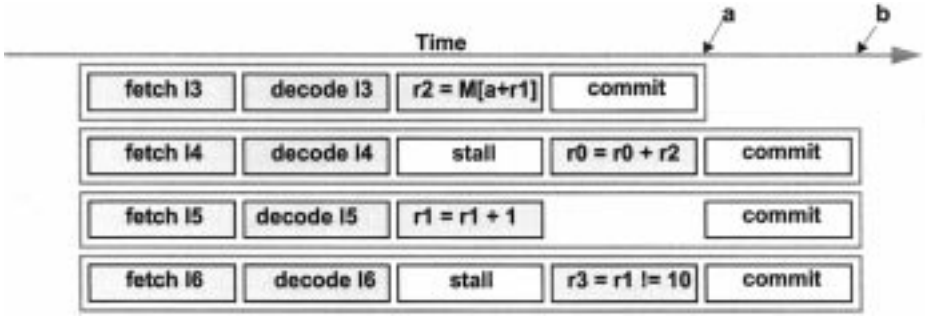
1) Superscalar Execution: Consider instructions I4 and I5 of Fig. 1(b). Pipelining will *partially* overlap their execution. *Superscalar execution* goes a step further *completely* overlapping their execution as shown in Fig. 3. In other words, I4 and I5 execute in *parallel*. In general, superscalar execution allows the parallel execution of adjacent instructions when such execution does not violate program semantics. For example, we cannot execute I6 in parallel with I5 as I6 needs the result of I5.

Superscalar execution and pipelining are orthogonal. We can build a superscalar engine that is not pipelined and vice versa and, of course, we can build a pipelined superscalar engine.

Superscalar execution requires additional hardware functionality. The first additional hardware requirement is a parallel decoder. Such a decoder needs to determine the dependence relationships amongst multiple instructions; this requires multiple comparisons between the source operands of an instruction and the destination operands of preceding instructions being considered for parallel execution. Superscalar execution also requires sufficient execution resources such as functional units, register read/write ports



(a)



(b)

Fig. 4. Out-of-order execution: (a) I3 and I5 execute in parallel. I4 and I6 have to wait for I3 and I5 respectively. Nevertheless, I4 and I6 also execute in parallel. (b) Introducing a commit stage to preserve sequential semantics.

and memory ports. Moreover, routing networks (or *bypass paths*) may be needed to route results among these resources. Additional information can be found in [19] and [23].

2) Out-of-Order Execution: As described, superscalar execution can execute two or more instructions in parallel only if they appear consecutively in the program order. Out-of-order execution relaxes this ordering constraint, thereby increasing concurrency [2], [63], [64].

In our code of Fig. 1(b), out-of-order execution would allow I5 to execute before I4 since I5 does not need a value produced by I4. A possible execution order facilitated by out-of-order execution is shown in Fig. 4. Instructions I3 through I6 are now fetched and decoded in parallel. Immediately after decode, I3 and I5 execute in parallel and I4 and I6 wait for I3 and I5, respectively. They can execute in parallel once I3 and I5 generate their results.

Out-of-order execution can result in a violation of the sequential semantics; therefore, additional effort is needed to ensure that the external appearance of sequential execution is maintained. For example, at point a in Fig. 4(a), the machine state is inconsistent: I3 and I5 have executed and I4, an instruction in between I3 and I5, has not. This discrepancy is of significance only if the machine's state is inspected at point a. Mechanisms to give the external appearance of sequential execution are discussed in Section II-B4.

3) Speculative Execution: So far, we focused on increasing the overlap between instructions whose execution is certain. Whether a block of instructions will be executed or not is determined by the outcome of control instructions, such as *conditional branch* instructions. In typical programs, a branch instruction occurs every fifth or sixth instruction. Thus, if we chose to overlap only instructions whose execution was certain, we would typically only be able to overlap the processing of five or six instructions at best. On the other hand, a superscalar processor with dual instruction issue, and with a ten-stage processing pipeline would need to overlap the processing of 20 instructions to keep the processing pipeline full. To obtain this additional concurrency, we are forced to look beyond branch instructions for additional instructions to process.

Speculative execution allows instructions beyond branches (i.e., instructions whose execution status is uncertain) to be processed in parallel with other instructions. The concept is simple: predict the outcome of the branch and speculatively process instructions from the predicted path. If the prediction was incorrect, make sure that the speculatively processed instructions are discarded and that they do not update the machine state. Speculation can be carried out past multiple branch instructions. (Later, in Section III-B, we will see how program behavior can be exploited to make accurate predictions on the outcomes of branches.)

The above form of speculative execution is also known as *control-speculative execution*, since the speculation is on the outcome of a control (branch) instruction. It overcomes ordering constraints due to *control dependences*, and is found on all modern high-performance microprocessors. Later, we discuss other forms of speculative execution that attempt to further increase concurrency by alleviating other ordering constraints.

The hardware required to support speculative execution is very similar to the hardware required to support out-of-order execution [21]–[23], [44], [54], [58]. The role of this hard-

ware is to maintain the external appearance of the sequential semantics of the program. (Now we are not only processing instructions out of program order, but are also processing instructions that might never have to be processed in a sequential execution.) We discuss this hardware in the next section. A point to note is that once hardware has been provided to support control-speculative execution, the hardware can typically support other forms of speculative execution with minimal changes.

4) Preserving Sequential Semantics: To understand how sequential semantics can be preserved while permitting a nonsequential instruction processing order, it is useful to assume that instruction processing now proceeds in three phases: *in-progress*, *completed*, and *committed*. An instruction is *committed* when it has completed execution, made state changes and these changes *are* visible to external observers. For example, in point a of Fig. 4(a), I3 is committed. A *completed* instruction has also executed and made changes, however, these changes are *not* visible to outside observers; they are visible only internally, e.g., to other instructions in progress. For example, in point a of Fig. 4(a), I5 is completed. The new value of r1 is visible to I6 (not to I4 or I3) but is not visible to outside observers. State changes made by completed instructions are conditional; they may get committed if execution is not interrupted (e.g., by an exception or by a misspeculation) or they may be discarded otherwise. Finally, *in-progress* instructions may have performed some actions but have not yet made any state changes. For example, in point a of Fig. 4(a), both I4 and I6 are in-progress.

By allowing instructions to be processed in any convenient order, we can improve concurrency, and consequently performance. Committing instructions in sequential order preserves sequential semantics. This is shown in Fig. 4(b). Notice that in point a, only I3 has committed, while in point b, all instructions have successfully committed. While I5 has completed at point a, any related state changes will be hidden to an outside observer and will be discarded if an exception occurs. Consequently, as far as an external observer is concerned, this machine adheres to sequential semantics.

The preceding discussion alludes to the existence of two distinct machine states. The first is the *architectural* state: the machine state as affected only by committed instructions. This state is visible to external observers and always adheres to sequential semantics. The second is the *speculative* state. This state includes all updates done by both committed and completed instructions.

Allowing Out-of-Order Completes While Forcing In-Order Commits: We now describe the actual mechanisms that allow out-of-order completes while forcing in-order commits. As instructions are fetched and decoded they are assigned a slot, in program order, at the tail of a queue-like structure, the reorder buffer. When instructions complete execution, they set a flag in their assigned slot. Committing instructions is done by inspecting the state of the instruction at the head of the *reorder buffer*. If the instruction at the head of the reorder buffer has completed execution, it can be committed. Thus, instructions are committed in program order. If an exception occurs, we simply discard the excepting instruction and all other instructions that follow it in the reorder buffer.

Maintaining Speculative and Architectural States: Updates to speculative state are done when an instruction completes, while updates to the architectural state are done when an instruction commits. When exceptions or other interruptions occur, it is necessary to discard the speculative state and resume execution using the architectural state. Therefore, what is needed is physical storage elements where the speculative and architectural values can reside, and a means for distinguishing between architectural and nonarchitectural values.

There are several ways of implementing this functionality. We limit our discussion to the register state. One option is to keep the speculative state in the *reorder buffer* [54], [58]. When an instruction completes, the value is placed in a field in the appropriate entry on the reorder buffer. When the instruction commits, the value is then written to the architectural register file. A second option is a *history-file* method [54]. Here a single machine state exists along with a log of recent, yet-to-be-committed changes. As instructions complete, they update the machine state. However, at the same time a log record is kept holding the previous value of the registers being updated. These records are maintained in program order similarly to what was done for the reorder buffer. When an instruction commits, no further action is necessary and the corresponding log record is discarded. If an exception occurs, the log records are used to restore the architectural machine state.

The above solutions have distinct storage space for architectural and nonarchitectural values. Another option is to have a combined storage space, a *physical register file*, and use *register mapping tables* to determine which physical registers contain architectural values, and which contain nonarchitectural value. This requires a scheme for *register renaming* [2], [17], [25], which we discuss next.

5) Register Renaming—Facilitating Out-of-Order and Speculative Execution: Register renaming was originally proposed to facilitate out-of-order execution, by overcoming artificial dependence constraints for a machine with a limited number of registers [2], [64]. Today, it is used to also facilitate speculative execution. We introduce this concept by describing its original purpose.

Consider the code snippet of Fig. 5 and inspect the various dependences that exist. Instruction I2 needs the value produced by I1, and stored in register r1. Therefore, a *true* or *read-after-write* dependence exists between these instructions and they must execute in order (in Section V-A, we will relax this ordering). Instruction I3 does not read any of the results produced by either I1 or I2. As described, out-of-order execution will not allow instruction I3 to execute before I1 or I2 because I3 overwrites the value of r1 produced by I1 and read by I2. That is, I3 has an *output* or *write-after-write* dependence with I1 and an *antidependence* or *write-after-read dependence* with I2. Collectively, output- and antidependences are referred to as *artificial dependences* to distinguish them from true dependences. They are also

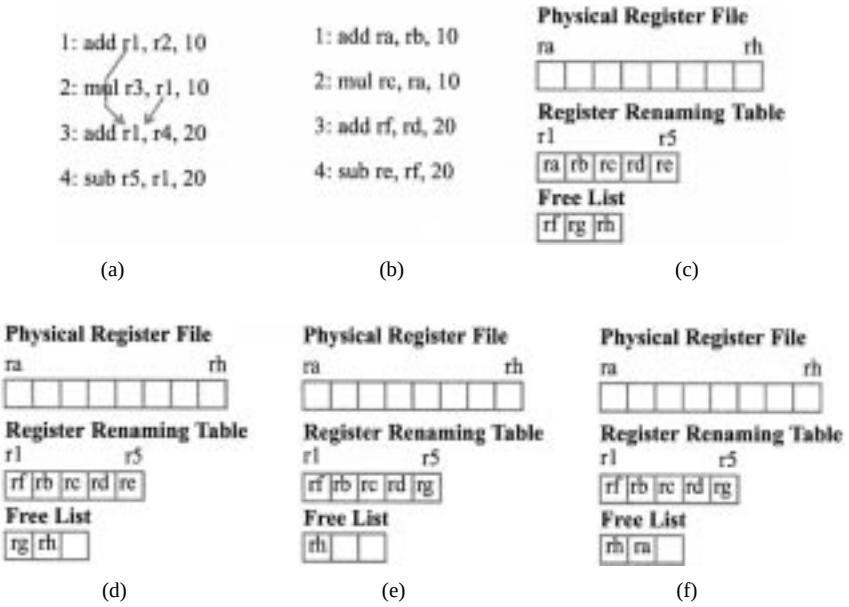


Fig. 5. Register renaming enhances our ability to extract instruction level parallelism. (a) Code with false dependences. Instruction 3 has an antidependence with instruction 2 and an output dependence with instruction 1 as it overwrites register r1. Without register renaming instructions will execute in order since there can be only one value for r1 at any given time. (b) If we had infinite register names, we could avoid false dependences. (c)–(f) Implementing register renaming. (c) After decoding instructions 1 and 2 registers r1 through r5 have been mapped to physical registers ra through re. (d) Decoding instruction 3. We first look at the current mapping for register r4. Then, we rename r1 to a physical register from the free list. (e) Decoding instruction 4. We find that now r1 is mapped to rf. Consequently, instruction 4 will read the r1 result produced by instruction 3 and not the one produced by instruction 1. Correctness is maintained as instruction 1 and 2 can communicate using physical register ra. We rename r5 to rg. (f) After committing instructions 1 and 2, we can release ra. No other instruction will ever need the value of r1 produced by instruction 1 as in the original program it would have been overwritten by instruction 3.

called *name dependences*, since they are a consequence of reusing the same storage name; no dependence and consequently ordering constraint, would have arisen had a different register name been used every time we wrote a new result [see Fig. 5(b)].

Register renaming maps register names to various storage locations and in the process it eliminates false dependences. In a processor with register renaming, there are (typically) more storage locations than there are architectural registers. When an instruction is decoded, a storage location is assigned to hold its result, and the destination register number is mapped to this storage location. At this point the target register has been *renamed* to the assigned storage location. Subsequent references to the target register are redirected to this storage location, until another instruction that writes to the same register is decoded. Then, the register is renamed to another storage location. A previously mapped storage location can be reused once the previous value is no longer required.

Speculative execution creates another application for register renaming. With speculative execution we have nonspeculative and speculative values residing in storage elements, some of which will be committed to the architectural state while others will not. In addition, multiple values bound to the same architectural register could be present, including nonspeculative and speculative values. Means are needed to direct a reference to a register to a particular storage element; this is the functionality provided by register renaming.

Many schemes for register renaming exist. A common implementation of this method comprises a *physical register file*, a *register renaming table* and a *free physical register list*, e.g., [17], [25]. The *physical register file* provides the storage for values. The *register renaming table* contains the current mapping of architectural register names to physical registers. The free list contains the names of those physical registers that are currently available for mapping. An example is shown in Fig. 5(c)–(e). Let us assume an initial state immediately after instructions I1 and I2 have been decoded. At this point, the architectural registers r1 through r5 are mapped to physical registers ra though re, while the free register list contains rf through rg. The register names for instructions I1 and I2 have been changed internally during decode. This was done by accessing the register renaming table to determine the current name assigned to each source register operand. Also, new names were allocated for registers r1 and r3, the targets of instructions I1 and I2 respectively. Upon decoding instruction I3, the physical registers for its two input registers are located by accessing the register rename table [Fig. 5(d)]. Then, a new name rf is allocated for its target register r1 [Fig. 5(d)]. As a result, when instruction I4 is decoded [Fig. 5(e)], it uses rf to communicate with instruction I3. Eventually, when instruction I2 commits, physical register ra is returned to the free register list so that it can be reused [Fig. 5(f)]. (In practice, the condition for returning a register to the free list is more stringent [25].)

Maintaining sequential semantics is done by extending the existing mechanisms deployed for out-of-order execution: we treat changes to the register rename table and the free list the same way we treat changes to register values. Register renaming decisions are *completed* during decode and are *committed* only when the corresponding instruction commits. This can be done using slight variants of the schemes described in Section II-B4.

6) Overview of Instruction Processing: Having understood the goals of overlapped, superscalar, out-of-order, and speculative instruction processing, and the means to accomplish the desired tasks, we now overview the processing of an instruction in a modern microprocessor. First, we *fetch* an instruction from memory. Then, we have to determine its input dependences and rename its target registers if any, a process typically referred to as *decode* or *dispatch*. Instructions are then sent to holding stations or *reservation stations* where they wait for their input operands to become available. An instruction whose input dependences have been satisfied is deemed *ready* for execution. Subject to resource availability (e.g., functional units or register file ports) a ready instruction is *selected* for execution. If selected, it can then commence execution, or *issue*. When an instruction completes execution, its result is broadcast to all waiting instructions. Instructions that are waiting for the value can copy it into the appropriate place. Checks are also made to see if the instruction can be *woken up*, i.e., whether all its operands are available, and thereby become ready for execution. After an instruction has been executed, it becomes a candidate for *retiring*, at which point it is *committed* in program order.

7) Overlapping Memory Instructions: So far, we have concentrated on instructions that operate on register operands, and we discussed a progression of techniques for increasing the overlap in processing such instructions. Load and store instructions have memory operands. Techniques to overlap the processing of loads and stores follow a similar progression to the techniques described for nonmemory instructions. However, since loads and stores are typically only a third as frequent as other instructions, the progression of techniques for loads and stores typically lags the techniques for other instructions.

Until the 1990s, most microprocessors executed load instructions serially even though the processing of nonmemory instructions was pipelined. Improving the overall instruction processing rate then demanded an improvement in the rate of processing loads and stores, and microprocessors of the early 1990s started to overlap the execution of loads and stores, in a manner similar to pipelining. To do so required the use of *nonblocking* or *lockup-free* caches [30], [60]. More recently, microprocessors have felt the need to execute multiple load/store operations simultaneously [1], [13], [20], similar to superscalar execution. Next-generation microprocessors are feeling the need to execute load and store operations out of order, and speculatively [8], [28], [40].

Techniques for the concurrent processing of load and store instructions, however, are more complicated than equivalent techniques for nonmemory instructions. This is because establishing dependence relationships amongst load and store instructions is more challenging: while register names are available in the instruction themselves, memory addresses are calculated at run time. Consequently, we cannot inspect loads and stores as they are fetched and decoded, and establish dependence relationships before executing any instructions. Rather, we have to wait until their addresses have been computed. Once addresses have been computed, associative searches (for matching addresses) are required to determine dependence relationships.

The above discussion reveals some of the difficulty in processing loads and stores. If we need to be sure that no dependence relationships are violated in a scheduling decision, an *address-based* scheduler needs as input the addresses being accessed by the all the loads and stores under consideration, and it needs to compare these addresses with one another. This not only requires complex search hardware, it delays the scheduling of loads until the addresses of all prior store instructions are known, e.g., [20], [39].

Recent approaches to scheduling loads propose the use of *data dependence speculation* to overcome the constraints for address-based schedulers, as described above. Rather than wait for all prior addresses to be know, i.e., for all *ambiguous* or *unknown memory dependences* to be resolved, a prediction is made as to whether a load is going to be dependent upon a prior store. If no dependence is predicted, the load is (speculatively) scheduled for execution. Later, checks are made to determine if the speculation was correct, with recovery actions initiated in case of incorrect speculation. As mentioned earlier, the basic mechanisms to support control speculation can also support data dependence speculation with little or no modifications. A review of relevant methods can be found in [39].

III. Exploiting Program Behavior

The techniques described in the previous section allow instruction processing to be performed in a certain manner. However, additional mechanisms are needed to support the basic instruction processing methods. We discuss several of them in this section. The mechanisms we discuss exploit the fact that typical programs do not behave randomly. There are uniformities and regularities that can be exploited.

The microarchitectural mechanisms that we discuss in this section can broadly be classified into two groups: 1) mechanisms to bridge the gap created by disparate rates of improvement for different parameters (e.g., logic and memory speeds) and 2) mechanisms to support or enhance previously described concurrent-processing techniques. Several of the mechanisms of this section and those of Section II are tightly coupled: individual mechanisms by themselves might not be of practical use in a given design.

A. Memory Hierarchies

1) Cache Memory: To goal of a memory hierarchy is to bridge the gap caused by the disparate rates of increase in speed between processor logic and memory.

Program memory demands have been increasing steadily: two decades ago, a few kilobytes of memory were sufficient

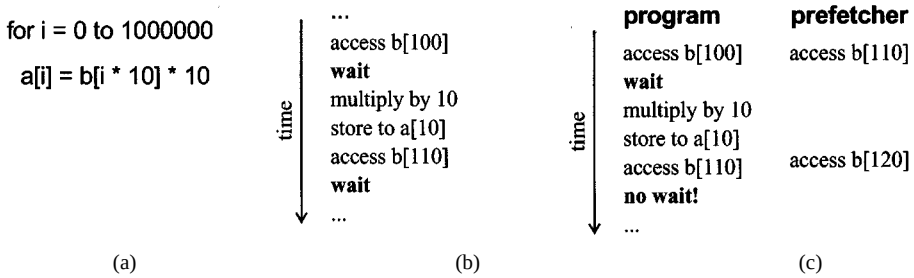


Fig. 6. (a) A code with regular array accesses. (b) Memory access timing without prefetching. Requests are not overlapped. (c) Prefetching requests the next array element while the previous request is still in progress. Request processing is overlapped and as a result overall throughput is increased. Notice that it still takes the same amount of time to fetch each individual array element.

to capture the memory footprint of most programs, today tens of megabytes are needed. This increase in memory demand is likely to continue and poses a continuous performance challenge for two reasons: First, programs access memory at least once for every instruction executed (to get the instruction itself) and twice if the instruction reads or writes memory data (about one in every three instructions in most programs). Second, sufficiently large memories tend to be very slow. Today, a typical access time for a DRAM memory is 40–100 processor clock cycles. This speed differential coupled with the frequent need to access memory can impair performance forcing even a processor with highly concurrent instruction processing to spend most of its time waiting for memory.

Sufficiently fast and large memories are unfortunately either not feasible or extremely expensive. However, fast but small memories are feasible and relatively inexpensive. *Caching* is a microarchitectural solution that approximates a large and fast memory using a small and fast one. A *cache* is a fast and small memory placed in between the processor and main memory. Initially, the cache is empty. After the processor references an address, a copy of the data as received from the slow main memory, is saved, or *cached* in the cache. If the processor references the same address later on, it would find that a copy resides in the cache (this is called a *cache hit*) and get the data much faster without having to access main memory.

Since the cache is much smaller than main memory, cached addresses are eventually replaced with newly referenced ones. For this reason, caches can reduce memory access latency only when a program references an address repeatedly and close enough in time so that it keeps finding it in the cache. Fortunately, most programs do exhibit this kind of behavior, which is referred to as *temporal locality*. In addition, many programs exhibit *spatial locality*. That is, once they reference an address A, they soon reference nearby addresses (e.g., A + 1 or A + 2). We can exploit this behavior by having cache entries (or blocks) hold multiple consecutive memory addresses. Accessing any address within the block results in the transfer of the whole block from memory. Accordingly, a cache is *prefetching* data in anticipation of future processor references.

Caches improve performance by reducing the *average* time required to access memory data; some memory references are sped up, others are not. They are also a *prediction-based* mechanism since they *guess*, albeit implicitly, that programs will exhibit both temporal and spatial locality. The basic concept was described in the 1960s [67], and there have been a number of improvements and alternate cache designs, e.g., [24], [52], [53].

What we have described is the simplest form of a *memory hierarchy* where a single level of caching exists in between the processor and main memory. To compensate for the ever increasing difference in processor and memory speeds, modern memory hierarchies consist of multiple caches, which are typically organized linearly. Caches closer to the processor tend to be faster and for that smaller; a modern microprocessor typically has two levels of cache memory on the processor chip. Being the part of the memory hierarchy closest to the processor, caches also provide a cost effective solution to supporting multiple simultaneous memory references [60].

2) Stream Buffers and Hardware-Based Prefetching: Basic caches perform implicit prefetching by fetching more data than that currently requested by the processor. Other more explicit forms of prefetching have been proposed and sometimes implemented in microprocessors. The basic idea is illustrated in Fig. 6. Part (a) shows a loop accessing the elements of array b[] using a linear progression. As shown in Fig. 6(b), without prefetching we first access one element, wait until it is received and then access the second one and so on. With prefetching as shown in Fig. 6(c), a mechanism generates speculative requests for subsequent array elements. For example, while the main program references b[100] the prefetching mechanism requests b[110], which is the next reference made to the data structure by the program. While individual requests take the same amount of time as before, overall performance is increased as multiple requests are overlapped.

Such techniques are very effective for numerical applications, applications that perform a large number of numerical computations over sizeable data structures which are typically arrays. Many of these applications exhibit fairly regular array accesses often accessing elements in a linear fashion (e.g., a[1], a[2], a[3] and so on), a program behavior that can be exploited. *Stream buffers* are hardware mechanisms that identify and exploit such access patterns to reduce memory latency [24]. They observe the addresses generated by the program while it is executing, looking for addresses that differ by a constant, or a *stride*. Upon identifying such patterns, stream buffers inject accesses to subsequent array elements using a simple calculation method such as "pre-

vious address + stride" [10 in our example of Fig. 6(a)], e.g., [7], [41]. If and when the program makes a reference to that address, it can be accessed, with a low latency, from the stream buffer, or the cache. Other forms of prefetching use heuristics such as prefetching multiple consecutive blocks on a miss [37], [53].

B. Branch Prediction

In Section II-B3, we motivated the need for control-speculative execution. Here, the outcome of a branch instruction is predicted using a scheme for *branch prediction*. Instructions are fetched from the predicted path, and executed speculatively while the branch execution progresses. The effectiveness of control-speculative execution is heavily influenced by the accuracy of the branch prediction, and the efficiency of the mechanisms to fetch instructions from the predicted path. Accordingly, these techniques have been studied extensively, and a considerable amount of hardware in a modern microprocessor is expended to achieve accurate branch prediction and instruction fetch.

Branch prediction can be broken into two subtasks: Predicting whether the branch will be taken, i.e., its *direction* and predicting its *target* address, or where the branch points to. These two issues are frequently studied separately, so we also separate our discussion of them.

1) Predicting the Direction of Branches: One class of branch direction prediction methods relies on *static* heuristics. For example, we could always predict branches as taken. This heuristic, though reasonably accurate, is not sufficient to fulfill the requirement for a modern branch predictor. Accordingly, several *dynamic* prediction methods have been developed and implemented.

Dynamic branch prediction methods observe branch outcomes as execution progresses and adapt their predictions accordingly [12], [42], [55], [62], [70], [71]. The simplest predictor of this kind uses a *last-outcome* approach and guesses that a branch will follow the same direction it did last time we encountered it. It works by recording branch directions (a single bit is sufficient) in a table indexed by the address of the branch instruction (the branch PC). Entries are allocated as branches execute and are updated every time the corresponding branch executes again. This table is typically referred to as a *branch prediction table*.

Further improvements exploit *temporal biases* in individual branch behavior [55]. For example, if the loop of Fig. 1(a) is executed multiple times, I7 will exhibit a T...TNT...TN... pattern, where "T...T" is a sequence of nine taken and N is a single not taken. The last-outcome predictor will mispredict twice every time the NT sequence appears. If we instead tried to identify the most frequent outcome (in this case taken), then only one misprediction will occur. A commonly used method uses a two-bit saturating counter per branch predictor entry. Values 0 and 1 represent a strong and a weak bias toward not-taken, while values 2 and 3 represent a weak and a strong bias toward taken. As branches execute we update these counters incrementing or decrementing them accordingly. Other automata besides counters have been used motivated by other commonly occurring branch constructs [31].

Another class of predictors relies on pattern-detection and association. A simple pattern-based predictor works as follows. A *history-register* records directions as branches are predicted. This is a shift register and it holds the last N branch directions (e.g., 0 for not taken and 1 for taken). Upon encountering a branch, its PC and the contents of the history register are used to index into a prediction table containing a single bit per entry. This bit indicates what happened last time the same combination of PC and past branch outcomes has been observed. In our loop example, such a predictor would associate the 9-bit history T...T with N (not taken). It would also associate any of the remaining nine histories with 8 T and one N with T (taken). There is a variety of pattern-based branch predictors each with its own advantages and disadvantages, e.g., [70]. For example, we could keep separate history registers per branch. This last point brings us to another important innovation in branch prediction, *correlation*. The observation here is that different branches exhibit correlated behavior. For example, if branch A is taken, then often branch B is also taken. Pattern-based predictors can exploit this phenomenon improving accuracy [12], [42], [62], [70], [71]. Pattern-based prediction can be generalized by including other pieces of relevant information that provide a strong indication of the direction of the branch they are associated with (e.g., the PC of preceding branches).

Some branch predictors work better than others for specific branches. Better overall results are possible by combining branch predictors [36]. For example, such a predictor may include a simple counter-based predictor, a pattern-based one and a predictor selector. To predict a branch, all three predictors are accessed in parallel. The predictor selector is used to select between the other two predictors. As execution progresses, the predictor selector is updated to point to the most accurate of the other two predictors on a per-branch basis.

2) Predicting the Target Address: If a branch is predicted to be taken, we have to also predict where it goes to. This is fairly straightforward for the majority of branches where the target address is always the same (PC + offset). We can predict such addresses using another PC-indexed table. Any time a branch executes, its target address is recorded in this *branch target buffer (BTB)*. When predicting a branch, we can obtain the target address from the BTB (if the corresponding entry is present). In another variation of the BTB scheme, in addition to the target address some of the target instructions are also stored within the BTB. Recall, that ultimately what we care about is fetching the target instructions.

Since function calls and returns are very common, special predictors have been developed for them. In these predictors, a short, hardware stack is kept. Upon encountering a call instruction, the return address is pushed onto the hardware stack. Upon encountering a return, the hardware stack is used to predict the return address [26].

The predictors we have described will fail for indirect branches having multiple target addresses (e.g., the "switch" statement in C or virtual function calls in C++). A number of

predictors for indirect branches have been proposed. They are based on generalizations of the schemes used for branch prediction, e.g., [5], [10], [27].

While modern branch predictors are highly accurate (often 90% or more), improving branch prediction accuracy is one of the most important problems in microarchitecture. As we move toward higher frequencies, longer pipelines and relatively slower main memories, it is desirable to extract more instruction level parallelism. This requires higher branch prediction accuracy. To understand the severity of the problem, let us consider a processor where up to 125 instructions can be active at any given point in time. Assuming a 90% branch prediction accuracy and that one in every five instructions is a branch, we can calculate the probability of filling up a window of this size to 0.9^{25} or just 0.07. (This is an approximate calculation but serves to illustrate our point.) This probability drops to 0.005 for a 250 instruction window. Highly accurate branch prediction is critical to increasing the usefulness of control-speculative execution. (In Section V, we will see other proposals for increasing concurrency that do not require all sequential branches to be predicted, as above.)

C. Trace Cache

In addition to high branch prediction accuracy, it is also desirable to predict multiple branches per cycle. Here is why: Modern processors may execute up to four instructions simultaneously. Processors capable of executing eight or more instructions per cycle are forthcoming. To sustain parallel execution of this degree, we need to fetch at least as many instructions per cycle. Assuming that on the average one in five instructions is a branch, we need to predict at least two branches per cycle. In practice, because branches are not necessarily spaced uniformly, we may need to predict more than that. A potentially complex and expensive solution to multiple branch prediction is to replicate or multiport existing predictors.

The *trace cache* offers a complete solution to predicting and fetching multiple instructions including branches per cycle [43], [45], [47]. It exploits the observed behavior that typically only a few paths through a program are actually active during an execution of the program.

The trace cache is a cache-like structure storing a sequence, or a *trace* of instructions as they have been fetched in the past. Consider the code of Fig. 7(a). Fig. 7(b) shows how a conventional branch predictor/cache organization would predict and fetch the sequence I1, I2, I4, I5, I6 (I2 is taken and I5 is not). Instruction fetch is limited by both branch count and memory layout. Even though we predicted that I2 is taken, we have to wait at least a cycle to fetch the target I4. This is because the latter is stored in a separate cache block. Fig. 7(c) shows how prediction and fetch would proceed with the trace cache. The first time this sequence is predicted, we access the conventional cache and pay the same penalties as in Fig. 7(b). However, as instructions commit, we store them in order into a trace cache entry. Next time around, all instructions become available in a single cycle from the trace cache. Passing this sequence as is to the rest of the processor is equivalent to predicting branches I2 and I5 as taken and not taken, respectively, and fetching

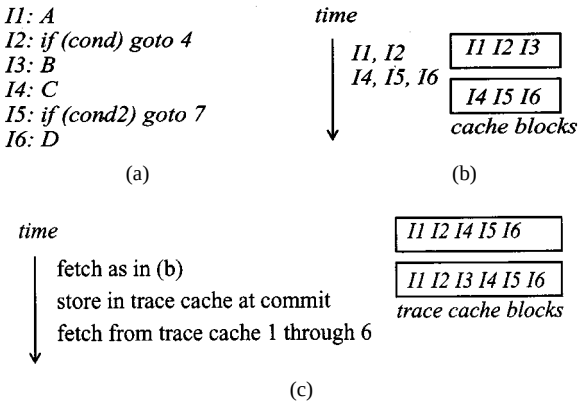


Fig. 7. Trace cache: predicting and fetching multibranch instruction sequences per cycle. (a) Code snippet. (b) Predicting and fetching in a conventional processor. We are limited to one prediction per cycle. Moreover, we are restricted to accesses within a single cache block. (c) Execution with a trace cache and trace cache entries.

all necessary instructions. In effect, we performed both multiple branch prediction and fetched multiple instructions. As shown in Fig. 7(c), it is possible to store instruction sequences corresponding to different paths through the code.

D. Memory Dependence Speculation and Prediction

As mentioned in Section II-B7, memory dependence speculation can be used to increase the concurrency in processing load and store instructions. Program behavior, the fact that a vast majority of loads do not exhibit dependences with store instructions that are in close temporal proximity, provides the rationale for *naive memory dependence* speculation. In this scheme, a load with unresolved dependences is always executed immediately, with the assumption that its execution will not violate any dependences.

As we attempt to extract more parallelism by having more instructions being processed concurrently, more loads experience memory dependences and naive speculation does not perform as well [40]. This suggests the need for more accurate *memory dependence prediction* techniques, which predict when a load is likely to be dependent on a prior store, and on which prior store is it dependent. Much better performance is possible when a load is synchronized with the store on which it depends. Fortunately, the memory dependence stream of typical program exhibits repetition, a behavior that memory dependence prediction exploits to predict and speculatively synchronize loads and stores [39].

IV. Putting It All Together: A Modern High-Performance Processor

To put it all together, Fig. 8(a) illustrates how instruction processing is conceptually carried out in a modern, high-performance processor. Instructions are fetched, decoded, and renamed in program order. At any given cycle, we may actually fetch or decode multiple instructions; many current processors fetch up to four instructions simultaneously. Branch prediction is used to predict the path through the code so that fetching can run-ahead of execution. After decode, instructions are issued for execution. An instruction is allowed to execute once its input data becomes available and provided

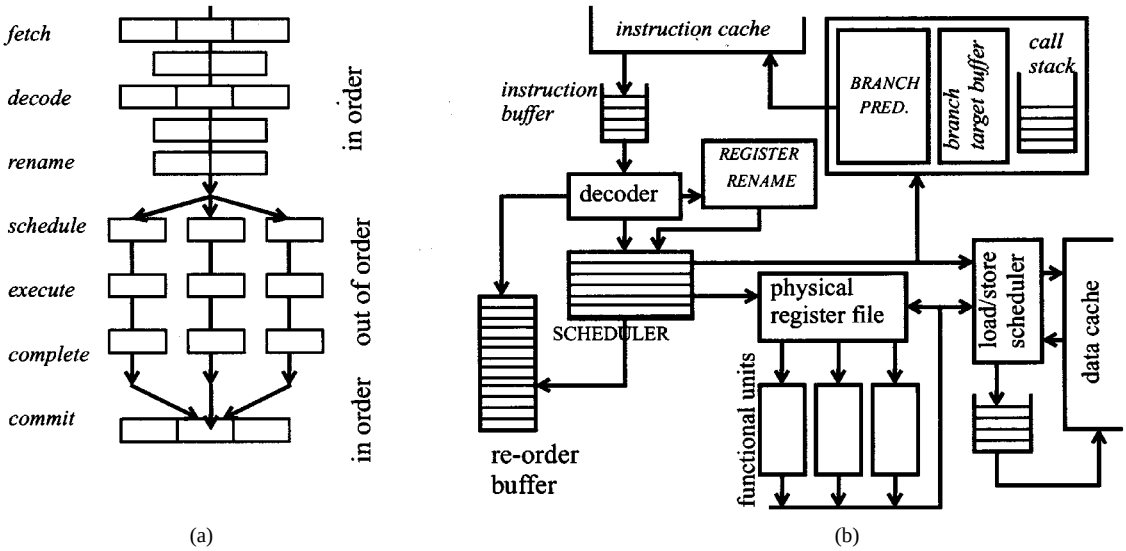


Fig. 8. Modern high-performance processor. (a) How execution progresses. Instructions are fetched, decoded, and renamed in program order. Multiple instructions are typically processed per cycle. Instructions are allowed to execute and complete in any permissible order based on operand and resource availability. Instructions commit in order. (b) Internal structure.

that sufficient execution resources are available. Once an instruction executes, it is allowed to complete. At the last step, instructions commit in program order.

Fig. 8(b) shows the underlying structures used to facilitate out-of-order execution. The front end, comprising an instruction cache and a branch prediction unit, is responsible for supplying instructions (a trace cache is used in some designs). As instructions are fetched, they are stored into the instruction buffer. The instruction buffer allows the front end to run ahead of the rest of the processor effectively hiding instruction fetch latencies. The decode and rename unit picks up instructions from this buffer and decodes and renames them appropriately. These instructions are then fed into the reorder buffer and the scheduler. At the same time, entries are also allocated in the load/store scheduler for loads and stores. Instructions are allowed to execute out of order. They access the physical register file using the physical names provided during decode and rename. Execution takes place in functional units. Multiple functional units typically exist to allow parallel execution of multiple instructions. Loads and stores are buffered in the load/store scheduler where they are scheduled to access the memory hierarchy. Stores are allowed to access memory only after they have committed; they are buffered into an intermediate buffer residing between the load/store scheduler and the data cache until they can commit. Instructions are committed in order using the reorder buffer. Maintaining the original program order in the reorder buffer also allows us to take corrective action when necessary, as is needed on a misspeculation (branch or data-dependence) or on an interrupt.

V. Future Microarchitectural Trends

So far, we have seen a variety of techniques that are mainly directed at the problem of achieving high performance in current microprocessors. Additional transistor resources were expended for microarchitectural mechanisms whose role was to extract, increase, and exploit concurrency in processing the instructions of a single, sequential program. This included mechanisms to allow instructions to be processed in arbitrary orders, allow instructions to be executed speculatively, and mechanisms that exploit program behavior to improve the accuracy of speculation. Transistor resources were also spent on memory hierarchies to bridge the gap created by disparate rates of improvement in logic and memory speeds.

Future microprocessors will be faced with new challenges. Performance will likely continue to be a challenge, with new parameters in the performance equation (e.g., the impact of increasing wire delays, e.g., [35], [38]). However, other challenges will arise, for example, power, design and verification, and reliability. Our expectation is that microarchitecture will be called upon to help solve these problems. We discuss some microarchitectural techniques that have been proposed in recent years, and are likely to be used commercially in the near future.

A. Data Value Speculation and Value Prediction

Earlier, we saw control speculation for control dependences and memory dependence speculation for overcoming ambiguous memory dependences. Data value speculation has been proposed as a way of overcoming true dependences, further increasing the available parallelism [15], [33].

With data value speculation, a prediction is made on the values of data items (the output of an instruction, or the inputs of an instruction). Two instructions that were dependent can be executed in parallel if the output of the first instruction (the input of the second) can be predicted with high accuracy. As with other forms of speculation, the speculation needs to be verified, and recovery actions initiated in case of incorrect speculation.

Surprisingly, it is possible to predict the values of many instructions with a reasonable accuracy. Empirical studies of

program behavior show that many instructions tend to produce the same value every time they execute. Others cycle through a small set of values following a regular pattern (e.g., 1, 2, 4, 1, 2, 4 and so on). And finally, some instructions follow well defined patterns such as increasing a value by a fixed amount (e.g., incrementing by 2 or 3). This observed program behavior is called *value locality* [32], [51], [57].

A number of value predictors have been proposed. Current value prediction schemes, however, do not achieve prediction accuracies high enough to obtain a benefit from value speculation; the costs of a misspeculation override any benefits from the additional concurrency. Should better value predictors be developed, value speculation is likely to find increasing use as a means to increase parallelism. More likely, limited applications of value prediction and speculation are likely to be developed for critical data dependencies where values are highly predictable.

B. Instruction Reuse

Another technique for exploiting value locality is *instruction reuse*. Unlike value speculation, instruction reuse is not a speculative technique. The observation here is that many instructions produce the same results repeatedly. By buffering the input and output values of such instructions, the output values of an instruction can be obtained via a table lookup, rather than by performing all the steps required to process the instruction. Simpler forms of reuse that do not require tracking of the actual values are possible [56]. Not performing all the steps of instruction processing can also benefit power consumption. Instruction reuse can also salvage some of the speculative work that is otherwise discarded on a branch misspeculation.

C. Microarchitectures With Multiple Program Sequencers

The microarchitectures that we have discussed so far have a single sequencer that fetches and processes instructions from a single instruction stream. As more transistor resources become available, a natural progression is to have microarchitectures that have multiple sequencers, and are capable of processing multiple instruction streams.

There is a spectrum of multiple-sequencer microarchitectures possible. At one extreme is a microarchitecture where multiple sequencers are added to the front end of the machine (instruction fetching, decoding, renaming), with the back end of the machine remaining mostly unchanged. Each instruction stream has its own set of front-end resources; the back-end resources are shared by all the instruction streams. At the other end of the spectrum is a microarchitecture that is a collection of multiple processing engines, one for each instruction stream. Here, the unit of replication increases from a functional unit capable of executing an instruction to a processing unit capable of processing a sequence of instructions. Other points in the spectrum, with varying degrees of sharing of hardware resources, are possible.

The first option is a straightforward extension to a modern superscalar processor, for example, the one shown in Fig. 8. It is also called a *simultaneous multithreaded (SMT)* processor [66], [68]. The second option implements a more traditional multiprocessor on a single chip, and can be viewed as a *chip multiprocessor (CMP)*. Logically, either microarchitecture appears to be a virtual multiprocessor. Microprocessors with an SMT are already on the design board, as are microprocessors with a CMP microarchitecture.

Because a CMP microarchitecture shares fewer resources than an SMT microarchitecture, it is likely to require more transistors than an SMT microarchitecture. However, the replicated, decentralized microarchitecture of a CMP is appealing from the viewpoint of future design problems: the use of replication alleviates the design and verification problems and decentralized microarchitectures with localized communication are more likely to be tolerant of wire delays than centralized microarchitectures that require more global communication.

We expect future microprocessors microarchitectures to have multiple sequencers, with the microarchitecture falling somewhere on the SMT-CMP spectrum, and shifting toward the CMP side as transistor budgets increase. Logically such a microarchitecture appears to be a *virtual multiprocessor*. This bring up an important question: what sequence of instructions should each processor of the virtual multiprocessor be responsible for processing? We address this issue next.

Using Multiple Sequencers to Improve Throughput: One approach to utilizing the multiple virtual processors is aimed at multiple, independently running programs or *threads*. The motivation here is that often processors are used in multiprogramming environments where multiple threads time-share the system. For example, microprocessors are typically used as processing nodes in a multiprocessor server.

A straightforward way of using the multiple virtual processors is to run a different program on each. Collectively, the processing throughput is increased, but the time taken to execute any individual program thread is not reduced. In fact, because of resource sharing, the time taken to execute a single program can actually increase.

Using Multiple Sequencers to Improve Single Program Performance: Since the time taken to execute a single program will continue to be important, researchers have been investigating techniques for using multiple virtual processors to speed up the execution of a single program.

Recall that a way to decrease the execution time of a program is to increase concurrency. An approach that has been used for several decades is to statically divide a program into multiple threads, i.e., to *parallelize* it. This approach is successful for regular, structured applications, but is of limited use for others. The main inhibitor to parallelization is the presence of *ambiguous dependences*. *Thread-level speculation* can be used to overcome these constraints and divide a program into *speculative threads*.

The *multiscalar processing model* is an example of using thread-level speculation to divide a sequential program into multiple speculative threads, also called *tasks* [59]. These tasks are then speculatively dispatched, based on control flow prediction, to the parallel processing units. Data is speculatively communicated among the tasks via the registers and a

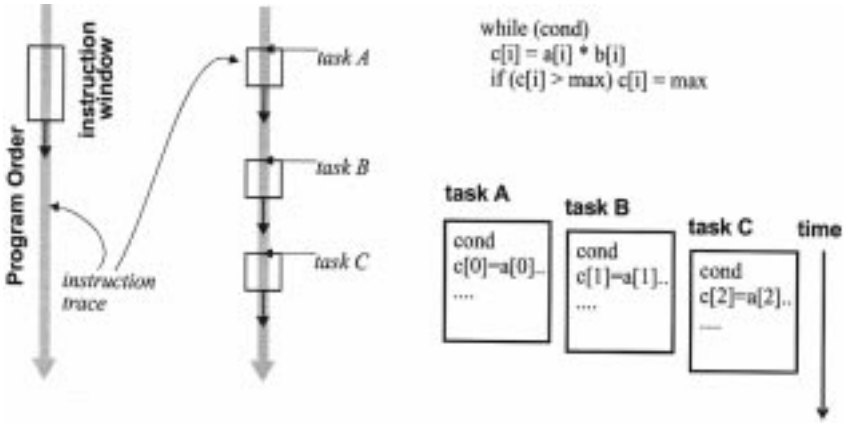


Fig. 9. The Multiscalar execution model. (a) Continuous, centralized instruction window (e.g., typical dynamically scheduled superscalar). (b) Multilscalar's way of building a large instruction window. (c) A loop. (d) How this loop may execute under the Multiscalar execution model.

specially designed memory mechanism, the *address resolution buffer (ARB)* [14] (or an alternative approach, the *speculative versioning cache* [16]). There is an implied sequential ordering between the tasks, which is inherent in the task creation, and is maintained by the microarchitecture. This allows the appearance of sequential execution to be recreated from the actual, speculatively parallel, execution.

An example of multiscalar execution is shown in Fig. 9. An iteration of the while loop comprises a task. A prediction is made that an iteration of the loop will be executed, and assigned to a processing unit (3 iterations have been spawned as tasks A, B, and C). These tasks are then executed in parallel as shown in Fig. 9(d). Notice that each iteration contains an if statement "c[i] $>$ max" that could be unpredictable. The corresponding branch would limit performance in a conventional processor as misspeculation (in the first iteration) would result in discarding all instructions that follow (the ones included in iterations 2 and 3). In the multiscalar model, however, such branches could be hidden inside each task restricting misspeculation recovery only within a task. Consequently, Multilscalar's model of extracting concurrency from a single program actually allows more concurrency that the superscalar model.

Two recent processors, Sun's MAJC [65] and NEC's Merlot [11], have implemented thread-level speculation concepts similar to those of the multiscalar model.

Another approach is the use of *helper threads*. The idea here is to spawn and run short threads in parallel with the main sequential thread to aid performance. For example, such threads might be prefetching data from memory, or they can be implementing a sophisticated branch prediction algorithm that is not economical or possible to implement in hardware. Such threads can be embedded statically in the program, or be dynamically extracted [6], [49], [48], [50], [61], [72].

D. Power-Aware Microarchitectural Techniques

Successive processor generations typically rely on more transistors and higher frequencies to deliver higher performance. Unfortunately, this also increases power requirements and density (i.e., power dissipated over the same chip area). Both requirements impose increasingly stringent constraints for modern microprocessors. With current projections, it is simply impossible to maintain the current performance/power/cost growth trend. Since the microarchitectural mechanisms are what consumes the power, they will have to be designed (or redesigned) to address power concerns. A number of recent research efforts are focusing on *power-aware* microarchitectural techniques. These are techniques that facilitate dynamic power versus performance tradeoffs or offer competitive performance with reduced power demands.

To understand the opportunities for power optimizations at the microarchitectural level, first we have to understand what are the sources of power dissipation in modern processors. There are two such sources: *dynamic* and *leakage* power.

Dynamic power dissipation occurs whenever a transistor or wire changes voltage (i.e., value). Dynamic power dissipation is proportional to product of the number of devices changing value, of the speed of these changes (i.e., operating frequency) and of the square of the voltage change. Reducing power dissipation is possible by reducing each of these factors. Power-aware microarchitectural techniques address the number of devices, and their switching speed, while taking performance into consideration. For example, judiciously disabling control speculation has been shown to reduce power dissipation with a minimal impact on performance [34].

Power is dissipated even when devices do not change values due to the imperfect nature of semiconductor-based transistors. This is the *leakage power*. In existing designs, leakage power is relatively small. However, as we move toward smaller transistors and lower voltages, leakage power increases rapidly [9]. Power-aware efforts in this area aim at cutting off power to devices while they are not being used. This is a challenging task as powering on and off devices requires some time and, hence, can severely impact performance [4], [18]. For example, it is possible to reduce leakage power in caches by deactivating parts of the cache

with a negligible impact on hit rate and performance [69]. Other microarchitectural techniques that trade fast (and leaky) transistors for more, but slower transistors are also likely to effective in addressing leakage power.

Additional information on current power versus performance trends can be found in [9].

E. Microarchitectural Techniques for Reliability

Reliability is also becoming an increasingly important consideration in microprocessor design. There are two forces at work. The first is design complexity. As designs becomes increasingly complex, validating their functionality under all possible combinations becomes virtually impossible. Second, as we move toward submicrometer designs and multigigahertz operating frequencies, transient errors become ever more probable. Transient errors are a side-effect of physical phenomena such radiation interference. Consequently, there is an increasing need for *fault-tolerant microarchitectures*. Fault tolerance in itself is a research field with a long history. However, a number of novel fault tolerance techniques, facilitated by other innovations in microarchitecture, have been proposed recently.

AR-SMT builds upon the SMT microarchitecture [46]. In AR-SMT, two copies of the same program are run simultaneously as two threads on the microprocessor core. The two threads are time-shifted by a few instructions, with one thread running ahead of the other. The results produced by the run-ahead thread are buffered and compared against the results of the other thread. If a transient error occurs, chances are that it will affect the execution of only one of the two threads and, thus, will be detected.

Another proposal, DIVA builds on concepts initially developed for value prediction [3]. A fast, unreliable processing core runs ahead of a second slower, but inherently more reliable processing core, producing results that may contain errors. The second processing core checks the validity of the results of the faster core. The slower core can keep up with the faster one as it uses the results produced by the faster core as value predictions, and can verify them in parallel without being constrained by dependences.

VI. Concluding Remarks

We have presented a variety of microarchitectural techniques that have facilitated the dramatic improvements in microprocessor performance. These techniques avail of the more plentiful transistors provided by semiconductor technology scaling to improve performance beyond that possible simply by improving transistor speeds. We broadly classified the techniques into two categories: techniques that increase the concurrency of instruction processing, and techniques that exploit regularities in program behavior.

For increasing the concurrency of instruction processing, we presented several techniques used in modern microprocessors: pipelining, superscalar execution, out-of-order execution and speculative execution. Many transistors in a modern microprocessor are used to implement these techniques.

Microarchitectural techniques that exploit program behavior either serve to address gaps created by the disparate scaling of different technologies, or to support techniques for increasing instruction processing concurrency. We presented memory hierarchies as an example of a technique with the former role: addressing the increasing gap in processor and memory speeds. Techniques serving the latter role included branch predictors, trace caches, and memory dependence predictors. Significant transistor resources in modern microprocessors are also used to implement these techniques.

The beauty of microarchitectural techniques lies in the fact that they are mostly transparent to entities outside the microprocessor hardware. This gives microprocessor architects the freedom to use the microarchitectural techniques of their choice, without having any material impact (other than providing increased performance, for example) to programmers.

As we move into the next decade, scaling in semiconductor technology will continue to provide more transistor resources with which to build microprocessors. However, microprocessor architects will be faced with newer challenges, including the gaps created by disparate rates of improvement in transistor and interconnection speeds, complexities of the design, debugging and verification processes, power consumption issues, as well as reliability issues, in addition to the continued demands for more performance. Many of the upcoming challenges and constraints will have to be dealt with innovative microarchitectural techniques. We presented several such techniques in this paper. These included data speculation and value prediction, instruction reuse, microarchitectures with multiple sequencers and thread-level speculation, power-aware microarchitectures and microarchitectural techniques for improving reliability and fault-tolerance. Additional novel microarchitectural techniques will, no doubt, be invented, as problems arise, and there is a need to address the problems in an efficient and transparent manner.

References

[1] D. Alpert and D. Avnon, "Architecture of the Pentium microprocessor," *IEEE Micro*, vol. 13, no. 3, pp. 11–21, 1993.

[2] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM system/360 model 91: Machine philosophy and instruction-handling," *IBM J. Res. Develop.*, pp. 8–24, Jan. 1967.

[3] T. Austin, "DIVA: A reliable substrate for deep submicron design," in *Proc. Annu. Int. Symp. Microarchitecture*, Dec. 1999, pp. 196–207.

[4] J. A. Butts and G. S. Sohi, "A static power model for architects," in *Proc. 33rd Annu. Int. Symp. Microarchitecture*, Dec. 2000, pp. 248–258.

[5] P.-Y. Chang, E. Hao, and Y. N. Patt, "Target prediction for indirect jumps," in *Proc. 24th Annu. Int. Symp. Computer Architecture*, June 1997, pp. 274–283.

[6] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt, "Simultaneous subordinate microthreading (SSMT)," in *Proc. 26th Int. Symp. Computer Architecture*, May 1999, pp. 186–195.

[7] T.-F. Chen and J.-L. Baer, "A performance study of software and hardware data prefetching schemes," in *Proc. 21st Annu. Int. Symp. Computer Architecture*, Apr. 1994, pp. 223–232.

[8] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proc. 25th Int. Symp. Computer Architecture*, June–July 1998, pp. 142–153.

[9] V. De and S. Borkar, "Technology and design challenges for low power and high performance," in *Proc. Int. Symp. Low Power Electronics and Design*, Aug. 1999, pp. 163–168.

[10] K. Driesen and U. Holzle, "Accurate indirect branch prediction," in *Proc. 25th Annu. Int. Symp. Computer Architecture*, June–July 1998, pp. 167–178.
[11] M. Edahiro, S. Matsushita, M. Yamashina, and N. Nishi, "A single-chip multiprocessor for smart terminals," *IEEE Micro*, pp. 12–20, Jul.–Aug. 2000.
[12] A. N. Eden and T. Mudge, "The YAGS branch prediction scheme," in *Proc. 31st Annu. Int. Symp. Microarchitecture*, Nov. 1998, pp. 69–77.
[13] J. H. Edmondson, P. I. Rubinfeld, P. J. Bannon, B. J. Benschneider, D. Bernstein, R. W. Castelino, E. M. Cooper, D. E. Dever, D. R. Donchin, T. C. Fischer, A. K. Jain, S. Mehta, J. E. Meyer, R. P. Preston, V. Rajagopalan, C. Somanathan, S. A. Taylor, and G. M. Wolrich, "Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor," *Dig. Tech. J.*, vol. 7, no. 1, 1995.
[14] M. Franklin and G. S. Sohi, "ARB: A hardware mechanism for dynamic memory disambiguation," *IEEE Trans. Comput.*, vol. 45, pp. 552–571, May 1996.
[15] F. Gabbay and A. Medelson, "Speculative execution based on value prediction," EE Dept., Technion-Israel Inst. Technology, Tech. Rep. TR-1080, Nov. 1996.
[16] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi, "Speculative versioning cache," in *Proc. Int. Symp. High Performance Computer Architecture*, Feb. 1998, pp. 195–205.
[17] G. F. Grohoski, "Machine organization of the IBM RISC system/6000 processor," *IBM J. Res. Develop.*, vol. 34, pp. 37–58, Jan. 1990.
[18] J. P. Halter and F. N. Najm, "A gate-level leakage power reduction method for ultra-low-power CMOS circuits," in *Proc. IEEE Custom Integrated Circuits Conf.*, May 1997, pp. 475–478.
[19] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach*, 2nd ed. San Mateo, CA: Morgan Kaufmann, 1996.
[20] D. Hunt, "Advanced performance features of the 64-bit PA-8000," in *Proc. COMPCON'95*, Mar. 1995, pp. 123–128.
[21] W. W. Hwu and Y. N. Patt, "HPSm, a high performance restricted data flow architecture having minimal functionality," in *Proc. 13th Int. Symp. Computer Architecture*, June 1986, pp. 297–307.
[22] ——, "Checkpoint repair for high-performance out-of-order execution machines," *IEEE Trans. Comput.*, vol. C-36, pp. 1496–1514, Dec. 1987.
[23] M. Johnson, *Superscalar Design*. Englewood Cliffs, NJ: Prentice-Hall, 1990.
[24] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. 17th Annu. Int. Symp. Computer Architecture*, May 1990, pp. 364–373.
[25] K. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, pp. 28–40, Apr. 1996.
[26] D. Kaeli and P. Emma, "Branch history table prediction of moving targets due to subroutine returns," in *Proc. 18th Int. Symp. Computer Architecture*, May 1991, pp. 34–42.
[27] J. Kalamatianos and D. R. Kaeli, "Predicting indirect branches via data compression," in *Proc. 31st Annu. Int. Symp. Microarchitecture*, Nov. 1998, pp. 270–281.
[28] R. E. Kessler, E. J. McLellan, and D. A. Webb, "The Alpha 21 264 architecture," in *Proc. Int. Conf. Computer Design*, Dec. 1998, pp. 90–95.
[29] P. M. Kogge, *The Architecture of Pipelined Computers*. New York: McGraw-Hill, 1981.
[30] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proc. 8th Int. Symp. Computer Architecture*, May 1981, pp. 81–87.
[31] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Comput.*, vol. 17, Jan. 1984.
[32] M. H. Lipasti, "Value locality and speculative execution," Ph.D. dissertation, Carnegie Mellon Univ., Pittsburgh, PA, Apr. 1997.
[33] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *Proc. 29th Annu. Int. Symp. Microarchitecture*, Dec. 1996, pp. 226–237.
[34] S. Manne, A. Klauser, and D. Grunwald, "Pipeline gating: Speculation control for energy reduction," in *Proc. 25th Annu. Int. Symp. Computer Architecture*, June–July 1998, pp. 132–141.
[35] D. Matzke, "Will physical scalability sabotage performance gains?," *Computer*, vol. 30, pp. 37–39, Sept. 1997.
[36] S. McFarling, "Combining branch predictors," Digital Equipment Corp., WRL, Tech. Rep. TN-36, June 1993.
[37] E. McLellan, "The Alpha AXP architecture and 21 064 processor," *IEEE Micro*, pp. 36–47, June 1993.
[38] J. D. Meindl and J. Davis, "Interconnect performance limits on gigascale integration (GSI)," *Mater. Chem. Phys.*, vol. 41, pp. 161–166, 1995.
[39] A. Moshovos, "Memory dependence prediction," Ph.D. dissertation, Univ. Wisconsin-Madison, Madison, WI, Dec. 1998.
[40] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Dynamic speculation and synchronization of data dependences," in *Proc. 24th Annu. Int. Symp. Computer Architecture*, June 1997, pp. 181–193.
[41] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *Proc. 21st Annu. Int. Symp. Computer Architecture*, Apr. 1994, pp. 24–33.
[42] S.-T. Pan, K. So, and J. T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," in *Proc. 5th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Oct. 1992, pp. 76–84.
[43] S. J. Patel, M. Evers, and Y. N. Patt, "Improving trace cache effectiveness with branch promotion and trace packing," in *Proc. 25th Annu. Int. Symp. Computer Architecture*, June–July 1998, pp. 262–271.
[44] Y. N. Patt, W. W. Hwu, and M. Shebanow, "HPS, a new microarchitecture: Rationale and introduction," in *Proc. 18th Annu. Workshop Microprogramming*, Pacific Grove, CA, Dec. 1985, pp. 103–108.
[45] A. Peleg and U. Weiser, "Dynamic flow instruction cache memory organized around trace segments independent of virtual address line," U.S. Patent 5 381 533 , 1994.
[46] E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," in *Proc. 29th Int. Symp. Fault-Tolerant Computing*, June 1999, pp. 84–91.
[47] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: A low latency approach to high bandwidth instruction fetching," in *Proc. 29th Annu. Int. Symp. Microarchitecture*, Dec. 1996, pp. 24–35.
[48] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *Proc. 8th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Oct. 1998, pp. 115–126.
[49] ——, "Improving virtual function call target prediction via dependence-based pre-computation," in *Proc. Int. Conf. Supercomputing*, June 1999, pp. 356–364.
[50] A. Roth and G. Sohi, "Speculative data-driven multithreading," in *Proc. 7th Int. Symp. High-Performance Architecture*, Jan. 2001.
[51] Y. Sazeides and J. E. Smith, "The predictability of data values," in *Proc. 30th Annu. Int. Symp. Microarchitecture*, Dec. 1997, pp. 248–258.
[52] A. Seznec, "A case for two-way skewed-associative caches," in *Proc. 20th Annu. Int. Symp. Computer Architecture*, May 1993, pp. 169–178.
[53] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, 1982.
[54] J. Smith and A. Pleszkun, "Implementing precise interrupts in pipelined processors," *IEEE Trans. Comput.*, vol. 37, pp. 562–573, May 1988.
[55] J. E. Smith, "A study of branch prediction strategies," in *Proc. 8th Int. Symp. Computer Architecture*, May 1981, pp. 135–148.
[56] A. Sodani and G. S. Sohi, "Dynamic instruction reuse," in *Proc. 24th Annu. Int. Symp. Computer Architecture*, June 1997, pp. 194–205.
[57] ——, "An empirical analysis of instruction repetition," in *18th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Oct. 1998, pp. 35–45.
[58] G. S. Sohi, "Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers," *IEEE Trans. Comput.*, vol. 39, pp. 349–359, Mar. 1990.
[59] G. S. Sohi, S. E. Breach, and T. Vijaykumar, "Multiscalar processors," in *Proc. 22nd Annu. Int. Symp. Computer Architecture*, June 1995, pp. 414–425.
[60] G. S. Sohi and M. Franklin, "High-bandwidth data memory systems for superscalar processors," in *Proc. Int. Symp. Architectural Support for Programming Languages and Operating Systems*, Apr. 1991, pp. 53–62.
[61] Y. Song and M. Dubois, "Assisted execution," Dept. EE Systems, Univ. Southern California, Tech. Rep. CENG-98-25, Oct. 1998.
[62] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt, "The agree predictor: A mechanism for reducing negative branch history interference," in *Proc. 24th Annu. Int. Symp. Computer Architecture*, June 1997, pp. 284–291.

[63] J. E. Thorton, "Parallel operation in the control data 6600," in *Proc. AFIPS Fall Joint Computer Conf.*, vol. 26, 1964, pp. 33–40.
[64] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Res. Develop.*, pp. 25–33, Jan. 1967.
[65] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse, "The MAJC architecture: A synthesis of parallelism and scalability," *IEEE Micro*, pp. 12–25, Nov.–Dec. 2000.
[66] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proc. 22nd Annu. Int. Symp. Computer Architecture*, June 1995, pp. 392–403.
[67] M. Wilkes, "Slave memories and dynamic storage allocation," *IEEE Trans. Electron. Comput.*, pp. 270–271, Apr. 1965.
[68] W. Yamamoto and M. Nemirovsky, "Increasing superscalar performance through multistreaming," in *Conf. Parallel Architectures and Compilation Techniques*, June 1995, pp. 49–58.
[69] S.-H. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. N. Vijaykumar, "An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance I-caches," in *Int. Symp. High-Performance Computer Architecture*, Jan. 2001.
[70] T.-Y. Yeh and Y. N. Patt, "A comprehensive instruction fetch mechanism for a processor supporting speculative execution," in *Proc. 25th Annu. Int. Symp. Microarchitecture*, Dec. 1992, pp. 129–139.
[71] C. Young, N. Gloy, and M. D. Smith, "A comparative analysis of schemes for correlated branch prediction," in *Proc. 22nd Annu. Int. Symp. Computer Architecture*, June 1995, pp. 276–286.
[72] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *Proc. 28th Int. Symp. Computer Architecture*, July 2001.

Andreas Moshovos (Member, IEEE) received the M.Sc. degree in computer science from the University of Crete, Greece, and the Ph.D. degree in computer sciences from the University of Wisconsin-Madison.

He is an Assistant Professor in the Electrical and Computer Engineering Department, University of Toronto, Toronto, Ontario, Canada. His research interests are in computer architecture and microarchitecture for high-performance, low-cost, and power-aware microprocessors.

Prof. Moshovos is a recipient of the NSF CAREER award. He is a Member of the ACM.

Gurindar S. Sohi (Member, IEEE) received the Ph.D. degree in electrical and computer engineering from the University of Illinois, Urbana-Champaign.

He is a Professor in the Computer Sciences and the Electrical and Computer Engineering Departments, University of Wisconsin-Madison. His research interests focus on architectural and microarchitectural techniques for high-performance microprocessors, including instruction-level parallelism, out-of-order execution with precise exceptions, nonblocking caches, speculative multithreading, and memory-dependence speculation.

Prof. Sohi is a Member of the ACM.