Speculative Data-Driven Sequencing for Imperative Programs

Amir Roth
Gurindar Sohi

Technical Report #1411

March 2000

Speculative Data-Driven Sequencing for Imperative Programs

Amir Roth and Gurindar S. Sohi
University of Wisconsin, Madison

Abstract

Conventional processing of imperative programs is a study in contrasts. High performance demands high levels of data-driven execution where instructions execute as soon as data dependences allow. However, the imperative program representation dictates control-driven sequencing, which allows only limited forms of data-driven execution. This paper describes a new technique, ***speculative data-driven sequencing*** *that, per its name, uses speculation to perform data-driven sequencing for selected computations of unmodified imperative programs. By isolating selected computations from their control-driven context, and freeing them from the sequential limitations of the instruction window, data-driven sequencing enhances a processor's ability to find parallelism and overlap latency.*

Prior work has provided the basic mechanisms for dynamically extracting computations from imperative programs and executing them in a data-driven manner. However, the data-driven computations executed in this context were limited to playing a strictly "advisory" role. With no way to inject results directly into the main computation, they were left to impact performance either by prefetching data or giving hints to the branch predictor. In this paper, we introduce a mechanism that allows work performed by a data-driven sequencing engine to be integrated directly into a control-driven program execution, allowing the technique to play a much more "active" role in processing. Our preliminary results indicate that speculative data-driven sequencing has the potential for significant performance gains on a variety of applications.

1 Introduction

Program processing consists of two activities: **sequencing** determines what instructions are to be processed next; **execution** carries out the operations indicated by these instructions. It is widely acknowledged that an important component of fast processing is **data-driven execution** in which instructions execute in an order determined by the availability of their input operands. With instructions executing as soon as they are able to, data-driven execution implies ideal levels of parallelism, latency tolerance, and performance. However, only instructions that have been sequenced to are candidates for execution and sequencing order is determined by the programming model and its representation. True data-driven execution requires **data-driven sequencing** which in turn requires a data-driven program representation.

The imperative or von Neumann programming model underlies the large majority of today's software. An imperative program representation describes sequencing order by means of a largely implicit, compact, and easy to understand **control path**. Compilers attempt to reorder instructions along this control path to approximate a data-driven sequencing order as much as possible. However, they are limited by the mandated generality of the produced code, code size constraints, imperfect dependence information, and separate compilation. As a result, the instruction order produced by **control-driven sequencing** often does not match the data-driven order as given by operand availability criteria. This inherent mismatch limits imperative program performance. When order mismatches occur, control-driven sequencing delays the execution of ready-to-execute computations in favor of not ready ones and, by doing so, impairs the processor's ability to overlap latencies. Control speculation techniques enable a limited degree of data-driven execution but not in a fundamental way.

In this paper, we present a novel processor framework that incorporates data-driven sequencing, and consequently true data-driven execution, into the heretofore purely control-driven regime of imperative processors. This new framework consists of two logical subsystems[1]. The control-driven (CD) subsystem is a conventional imperative processor. The speculative data-driven (SDD) subsystem executes selected computations on behalf of the CD subsystem. The SDD subsystem requires a data-driven computation representation. Traditionally, this information had to be supplied at the architectural interface [6, 8, 3, 14]. However, in the CD/SDD framework, it is constructed speculatively and micro-architecturally from the imperative representation allowing data-driven sequencing to be applied to **imperative program executables**.

Mechanisms for dynamically extracting data-driven sequencing information from imperative programs and acting on it have been developed in earlier work [15, 16]. However, in that work, the SDD subsystem was relegated to playing only an "advisory" role in program execution. No attempt was made to merge the work performed by data-driven computations back into the main imperative computation, leaving SDD to impact performance by either acting as an intelligent prefetching engine or by supplying hints to the branch predictor. In this paper, we take that additional step and provide a mechanism that allows the results of data-driven computations to be seamlessly integrated into a control-driven execution without the need to repeat any of the work. The enabling observation is that since speculative data-driven sequencing initially derives from a totally ordered representation, it should be possible to maintain enough information along with the extracted computations to allow this total ordering to be reconstructed. **Integration** allows data-driven sequencing to play a much more "active" role in program execution.

Although we expand the role of data-driven sequencing, our preliminary evaluation of the new framework still focuses on the pre-execution of carefully selected computations. Specifically, our first experiment will attempt to perform data-driven pre-execution of computations that result in data cache misses. In most of our experiments, fewer than 5% of the dynamic instructions in the program are executed in the data-driven subsystem. Our results show that even this initial division of labor yields significant performance benefits.

The structure of the paper is as follows. Section 2 expands on the motivation and background given in the introduction. Section 3 presents the CD/SDD processor organization and describes the novel features and interface mechanisms that allow work from speculative data driven computations to be integrated directly into a control-driven execution. Section 4 describes a particular implementation of this new organization using a modified explicit data-flow core to perform data driven sequencing and execution. Initial performance results are given in Section 5. The last two sections present related work and our conclusions.

1. We use the names CD, control-driven, SDD and data-driven when referring to computations and values with the natural implications (e.g. a data-driven value is produced by instruction executing in the data-driven subsystem).

2 Motivating Example

A program is a collection of computations. A computation is a dynamic group of dependent instructions that ultimately produces one value. The instructions in a program obey a **partial order**: instructions within a computation are ordered by data dependences, instructions in different computations are logically unordered. Imperative programs superimpose a **total order** on top of this partial order with a point in the dynamic total order encapsulating the architected state of all computations in the program. An architected total order is a nice property for programmers. It makes per-instruction program state unique and gives program execution a **repeatability** property that is important in software and hardware development. It also allows the use of programming languages that permit statically ambiguous dependences. The disadvantage of an architectural total order is that it interleaves computations with one another so that they cannot be fully disentangled, either statically or dynamically. Computation interleaving degrades performance by forcing computations to execute at the pace of the slowest computation in their sequential proximity.

We illustrate using an example. The program fragment in Figure 1(a) is a simplified version of the core loop in the program *em3d*. Each loop iteration incurs three cache misses shown in bold: the first access to the element (*I1*), and the accesses to the values in each of the neighbor elements (*I5* and *I6*). The latencies of same-iteration instances of the *I5* and *I6* can be overlapped with one another, but not with *I1* due to data dependences. Inter-iteration overlapping is possible provided the recurrence *I11* can be broken (by either prediction or scheduling). The dynamic code sequences have been scheduled (statically or dynamically) to take advantage of as much intra-loop overlapping as possible assuming a latency of four issue slots per cache miss. We study the execution of two iterations.

The superscalar machine in part (b) of the figure cannot exploit inter-iteration overlapping because its contiguous window prevents it from sequencing to *I11* instruction at the end of the first iteration while code from the beginning of that iteration is still executing. Only latencies associated with instructions less than one window's worth apart can be overlapped. The execution shown in part (c) of the figure employs a technique called speculative-multithreading [2, 17]. The instruction stream is split into dynamically contiguous threads which are speculatively executed in parallel. Special hardware uses the logical ordering among the threads to enforce control and data dependences and maintain the illusion of sequential execution. Speculative multithreading succeeds when it can fully encapsulate one or several computations within a thread, allowing it to be essentially decoupled from the rest of the program. However conventional speculative threads are internally control-driven and hence are selected primarily by their control relationship to their surrounding context. In our example, speculative threads were partitioned along outer iteration boundaries. Although sequencing is less constrained here (both iterations can be sequenced in parallel), the end-of-iteration pointer recurrence *I11* still prevents cross-iteration latency overlapping[1].

The computation required to generate the cache misses in given iteration is only five instructions long: *I11* (from the previous iteration) and *I1*, *I2*, *I5* and *I6* and shown in Figure 1(a) as well. Operand-wise, this computation is ready to

1. In this case, the code can be statically restructured to move the *nd = nd->next* recurrence to the top of the loop. In general, the memory aliasing analysis required to guarantee that such a transformation is safe is not possible.

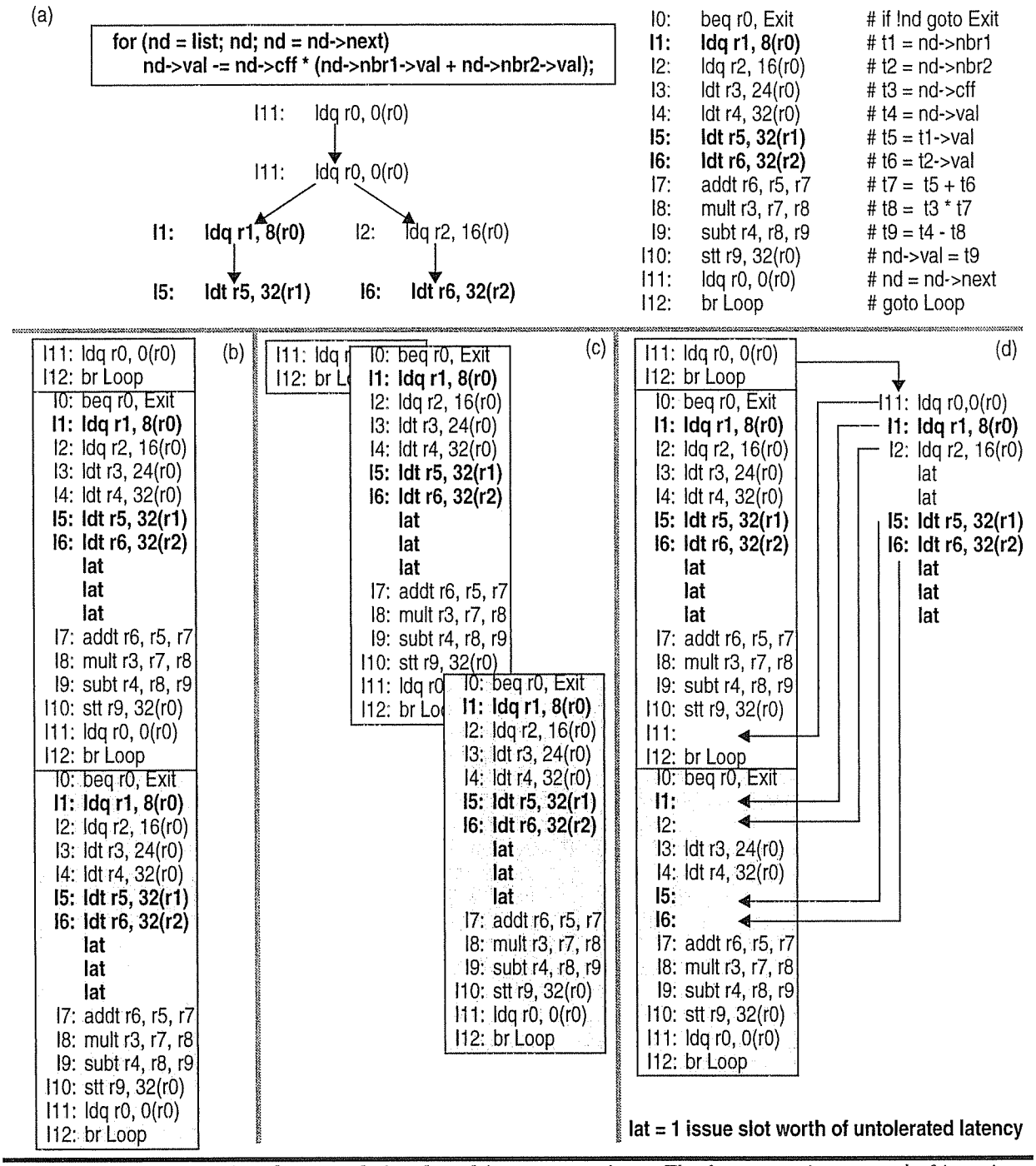

FIGURE 1. Motivation for speculative data-driven sequencing. *The loop contains an end-of-iteration recurrence that prevents latencies from different iterations from being overlapped. Three different executions of two iterations each (plus the end of a previous iteration) are shown. Each iteration is boxed, with the target iteration shaded.* ***lat*** *represents one issue slot of untolerated latency. Execution:(b) Superscalar. (c) Speculative multithreaded (d) Superscalar augmented with speculative data driven sequencing.*

execute at the **beginning** of the previous iteration. Where the control-driven approaches fail is in their inability to separate this performance critical computation from its control-driven context so that it can execute in a timely manner. Enter speculative data driven sequencing which removes the control path contiguity requirement on computation sequencing. Performance critical computations can now be sequenced and executed as soon as data dependences

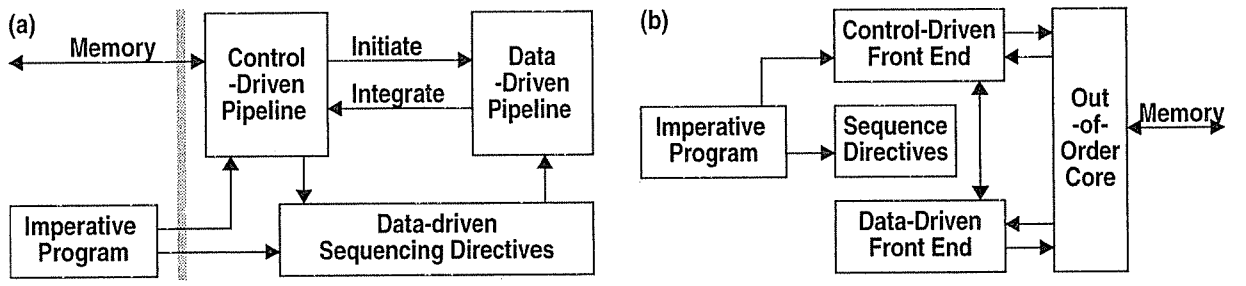

FIGURE 2. CD/SDD processor organization. (a) Logical organization with two distinct subsystems. (b) One possible physical organization with separate subsystem front ends sharing an execution core.

allow without sequencing the non-critical computations with which they are interleaved. The pre-computed results are then merged seamlessly back into the control-driven context without the need for re-execution. By allowing inherently fast computations to be separated from the slow computations around them and to execute at their natural pace, greater opportunity for exposing and exploiting parallelism is achieved.

An execution that exploits speculative data-driven sequencing is shown in part (d). First, the critical computation (*I11*, *I1*, *I2*, *I5*, *I6*) is identified and annotated with data driven sequencing directives. During conventional control-driven execution, these directives are used to dynamically extract the computation from its control-driven context and to sequence to it and through it in a data-driven manner. The end result is that *I11* executes at the beginning of the first iteration and the miss generating portion of the second iteration to be overlapped with code from the first iteration. During control-driven execution of the second iteration, the targeted cache misses are no longer observed.

3 Speculative Data Driven Sequencing

CD/SDD is a new processor organization that enables speculative data-driven sequencing for selected computations of imperative programs. Logically, a CD/SDD processor consists of two cooperating subsystems as shown in Figure 2. The control-driven (CD) subsystem is a generic imperative processor that executes an imperative executable and interfaces conventional memory; its structure is well understood. The outside world observes the processor only through the CD subsystem, its output determines correctness and its throughput defines performance. In contrast, the speculative data-driven (SDD) subsystem operates behind the scenes on behalf of its CD counterpart. Its precise structure and mechanics are not well understood, but neither are they fundamentally important. Operationally, the SDD subsystem should be capable of sequencing through computations as specified by some set of micro-architectural directives, and of naming and storing any computed results in a way that will allow their future integration into a control-driven execution. In this section, we describe the internal architecture of a CD/SDD processor - the interface between the two subsystems - abstracting the implementation of the SDD component (we provide a sample implementation in the next section). We show how an appropriate choice of interface primitives can facilitate the construction of a simple result integration algorithm.

3.1 The CD/SDD Interface

There are three aspects to the CD/SDD interface. The **architecture** defines the communication channels and the responsibilities of each party. The **language** defines the meanings of the individual pieces of data and control exchanged. The **format** defines the structure and content of the messages. For now, we are concerned with the language and architecture of the interface. The format may change depending on the particular implementation of the SDD component.

Architecturally, the centerpiece of the CDS/SDD interface is an internal data-driven representation of selected computations in the program. The control-driven processor constructs this representation by micro-architecturally annotating the imperative program executable with data-driven sequencing directives. The data-driven subsystem uses this representation in lieu of an executable to guide sequencing and execution. The CD "master" also communicates with its SDD "slave" using two more explicit channels. Along one, it **initiates** data-driven execution of selected computations and supplies the necessary input operands. Along the other, it **integrates** the values produced by these computations back into the control-driven execution. Each component will be described in detail later in the section.

Its "master" status also means that the control-driven subsystem defines the language of the interface. The primary reason for this is our desire to modify the control-driven subsystem as little as possible, both for simplicity of discussion and for practical reasons. That said, labeling data-driven results using control-driven terminology facilitates the integration process. Control-driven naming conventions apply to both instructions and data. For an instruction to be integrated into the control-driven side, it must correspond to some instruction in the imperative executable and must be labeled so that the control-driven processor can recognize this correspondence. Referring to an instruction by its address (PC) accomplishes this. Similarly, any result produced in the SDD subsystem must be stored in a location accessible to the CD subsystem and referred to by the control-driven name for that location. Following control-driven values, data-driven values must reside in physical registers or in memory and, at least over the initiation and integration channels, be referred to by a physical register number or a memory address.

3.2 Annotation Representation and Creation

The annotation mechanism refers both to the representation of the data-driven sequencing directives for those computations that will execute in the SDD subsystem, as well as to the apparatus that constructs them. Although both aspects are ultimately important, neither plays a major role in this paper. The annotation structure represents the instructions in the chosen computations and their data-driven sequencing order. We have already established that, at the language level, instructions must be represented by their PCs. However, the sequence order representation depends on the implementation of the SDD subsystem and, accordingly, may be either explicit (each instruction names its successors) or implicit (the instructions are listed in data-driven order). The name of the SDD subsystem refers to its logical function, not necessarily to the sequencing mechanism it uses. To wit, any method of sequencing through a single computation yields a data-driven order (the instructions would be sequenced in data-driven order and without delays caused by instructions from other computations). The SDD engine we present in this paper uses a

data-driven sequencing mechanism and an explicit sequence order representation. This choice was made to facilitate exposition and to allow us to draw clearer parallels with previous works, not due to fundamental requirements.

The annotation generation mechanism, which implements the computation selection policy, is inarguably the most important and delicate piece of the CD/SDD framework. However, its mechanics are both logically and physically divorced from those of the rest of the system. The rest of the paper will assume "magically" created annotations. In the real world, data-driven sequencing annotations can be generated by special hardware that observes a control-driven execution, loaded into the annotation table in software, or extracted using simplified hardware with the help of software hints. We predict that due to the importance of computation selection, software will play some role in any successful implementation. However, prior work has shown the viability of hardware-only mechanisms [15, 16].

3.3 Computation Initiation

The initiation mechanism triggers the execution of data-driven computations with the appropriate input operands. The mechanics of initiation will be discussed in a later section since they mostly involve implementation details of the SDD subsystem. However, **initiation policy** bears discussion here. Specifically, computations should be represented and interpreted in such a way that they can only be initiated from selected points. Instructions interior to a computation should not be allowed to initiate SDD action. Failure to make these distinctions will result in much repeated and wasted effort on the part of the SDD subsystem. Consider our five-instruction miss-generating computation from Figure 1. Since only the first instruction, *I11*, is allowed to initiate execution, then for each computation instance each instruction in the computation will execute once in SDD for a total of five instructions. However, if every instruction were allowed to initiate execution, then *I5* and *I6* will execute twice per computation instance, having been initiated once indirectly by *I11*, and once directly by *I1* and *I2*, respectively. In this case, the data-driven subsystem would execute seven instruction per-computation instance, only five of which would be useful.

3.4 Result Integration

Integration is the process of re-inserting a data-driven computation into the control-driven totally-ordered context from which it was originally extracted. Next to the mechanism that selects and represents computations, the integration mechanism is the most important of the CD/SDD interface components. Without it, the data-driven subsystem can impact the performance of its control-driven master (and hence the entire system) only indirectly (e.g. by prefetching data). An efficient mechanism allows data-driven work to benefit control-driven execution directly and without the need for re-execution (e.g. in Figure 1(d), the five instructions of the miss-generating computation in are not re-executed in the control-driven subsystem). We use the fact that the data-driven results are stored in physical registers (a property designed into CD/SDD specifically for this purpose) to implement integration as a simple addition to the control-driven register renaming process.

3.4.1 Integration Algorithm

A control-driven processor tracks the results of the many computations in the program using a register rename map (or just rename map) which translates architectural names (logical registers) into physical storage locations (physical registers). The rename map is a structure with control-driven semantics; an entry in the rename map implies that the corresponding physical register holds an active temporary value for some current control-driven computation. As it performs data-driven execution of a computation, the data-driven subsystem saves the result of each instruction in a physical register it "steals" from the free pool. However, since an instruction executed in SDD is not part of the sequential program, a stolen physical register is not pointed to by a logical register in the control-driven rename map. The observation that simplifies integration is that a pointer in the register map that is the **only thing** that distinguishes a control-driven instruction from a data-driven one. To integrate a data-driven instruction into a control-driven execution, we simply "convert" it into a control-driven instruction by setting the appropriate entry in the rename map to point to its physical register.

The actual integration algorithm is quite simple. As each control-driven instruction is being renamed, integration logic decides whether that same instruction has previously been executed in the SDD subsystem. If so, the data-driven instruction is effectively integrated by setting the rename map entry for the output of the control-driven instruction to the corresponding data-driven physical register. Otherwise, the output of the control-driven instruction is mapped to a newly allocated physical register, as usual. The decision procedure itself relies on the input physical register names of the control-driven instruction. To be integrated, a data-driven instruction must have executed with the same physical register inputs as it would have used in a control-driven context. The SDD subsystem facilitates this logic by maintains a table of recent in-flight and completed data-driven operations. Each table record consists of the instruction's PC (also used to index the table) and its input and output physical register numbers.

The new renaming algorithm (modified to include integration) proceeds as follows. First, the input logical registers of the control-driven instruction are translated to physical registers using the rename map as usual. In parallel, the instruction PC is used to probe the SDD result table. The decision logic then compares the physical register names obtained by control-driven renaming and those from the SDD table entry. Integration succeeds if the names match. Integration proceeds "recursively", instruction by instruction. The first instruction in a computation passes the integration test by virtue of having as its inputs already-mapped physical registers that were "given" to it by the control-driven execution. Subsequent instructions are integrated because their input registers are the outputs of data-driven instructions that have previously been integrated. Figure 3 shows the integration process schematically for two dependent instructions. Like conventional register renaming, register renaming with integration is a sequential process, but permits a parallel prefix formulation to allow multiple simultaneous operations.

3.4.2 Integrating Loads

For most types of instructions, integration based on physical registers is perfectly safe. The combination of operation (denoted by the PC) and input values (denoted by physical register names) is enough to unambiguously specify the

output. Loads are the exception to this rule. A load cannot be blindly integrated into a control-driven execution because a conflicting control-driven store may have occurred between its data-driven-execution and its integration. There is no way for the data-driven subsystem to know about these stores since it has no notion of computations other than the ones it is charged with executing, much less the total ordering between them. Integrating loads using physical register criteria only (addresses are not available at register rename time) allows integration of possibly incorrect values (**mis-integrations**) and jeopardizes program correctness. To be sure, load mis-integrations are extremely rare. However, even low frequency events must be **handled correctly** when they do occur.

The catch-all solution to ensuring correctness is to "re-execute" every integrated load in the control-driven subsystem. A mis-integration is detected when the initial data-driven value and the re-executed control-driven value differ, at which point all instructions dependent on the load must be replayed. Support for such replays already exists in most out-of-order processor schedulers [12]. Fortunately, re-executing integrated loads does not mean that every integrated load must access the data cache twice (once in each subsystem). Two tricks can be used to make double cache access the exception rather than the rule.

The first technique exploits the catch-all control-driven re-execution to forgo data-driven execution altogether for some loads. Previous research has shown that memory communication between stores and loads is highly regimented and predictable to extremely high levels [13]. Our framework can exploit this regularity and annotate data dependences between stores and loads directly. In the SDD subsystem, these dependence arcs are executed as register moves, without cache access or even address calculation. We call this active form of memory dependence speculation **speculative data-driven bypassing**. In addition to preventing double cache accesses, it often significantly shortens the path through a computation. Speculative data-driven bypassing can be performed knowing that the re-execution mechanism will detect and recover from any mistakes. This is a good time to explicitly mention stores do not execute per-se in the data-driven subsystem. In the data-driven world, stores exist for the sole purpose of forwarding their values to loads. Integrated stores are committed to the data-cache at retirement, as usual.

The second technique applies to non-bypassed loads and is the micro-architectural equivalent of the architectural load speculation mechanism of the IA64 architecture, the Advanced Load Address Table (ALAT) [11]. The ALAT is a small cache of addresses. Data-driven loads deposit their addresses in the ALAT while control driven stores remove matching addresses from the ALAT. Once an integrated load is disambiguated in the control-driven subsystem, it simply checks the ALAT for a matching address. A match means that no conflicting store occurred since the time the load was executed in the data-driven subsystem and that the load need not access the data cache again. This mechanism also forces bypassed loads, which do not deposit addresses in the ALAT simply because they do not compute addresses, to "re-execute" in the control-driven subsystem, ensuring that every load actually executes at least once.

3.5 Initiation/Integration Example

We use our miss-generating computation from Figure 1 to illustrate the processes of initiation and integration as viewed by the control-driven processor (a view from the data-driven side will be given in the next section). Figure 3

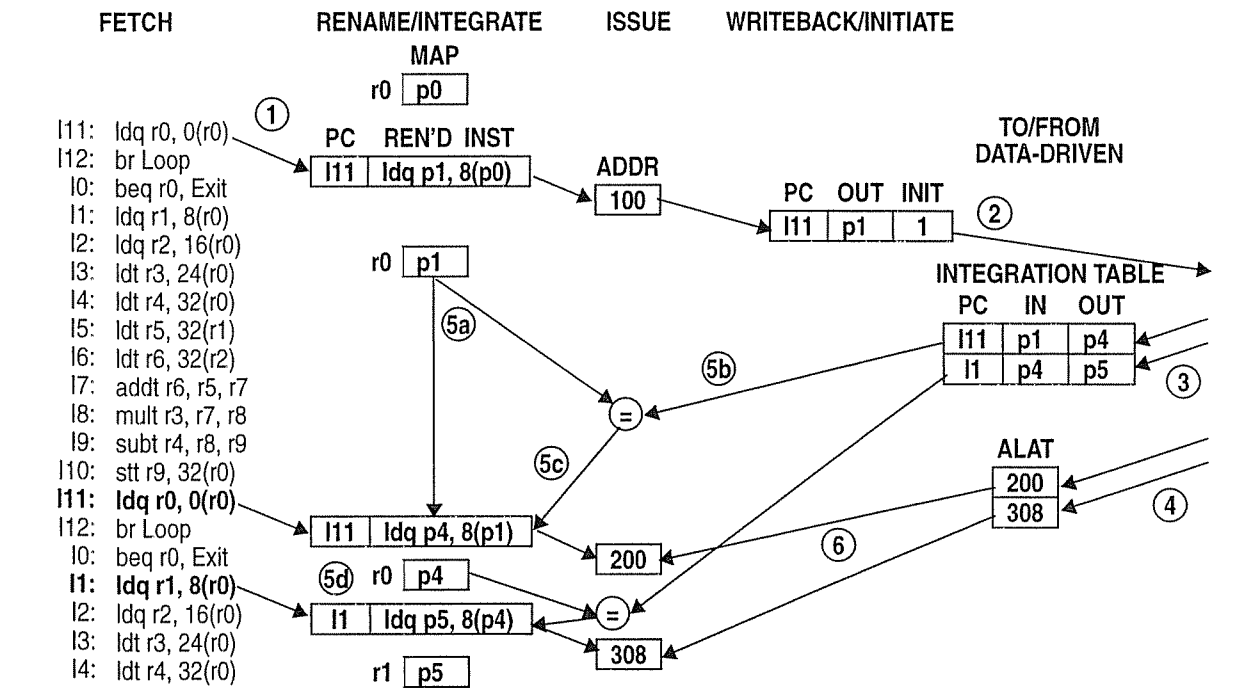

FIGURE 3. Working example: the control-driven side of initiation and integration. *The miss-generating computation from Figure 1 is initiated and the first two instructions (I11 and I1) are integrated.*

shows the sequence of actions (circled numbers) for initiating and integrating the first two instructions in the computation, *I11* and *I1*. We begin as instruction *I11* from the previous iteration is fetched and its target *r0* allocated physical register *p1* (1). When this instruction completes, the control driven processor sends the triple *<I11, p1, 1>* to the data-driven subsystem initiating the execution of the computation with input *p1* (2). The last bit in the triple signals a processor initiation request and allows us to implement the selective initiation policy we described in Section 3.3. In the data-driven subsystem, *I11* is first executed with input *p1* and output *p4* stolen from the free list. *I1* is then executed with *p4* as input and a similarly stolen *p5* as output. Both executions are recorded in the integration table (3) and the respective addresses of each data-driven load, 200 and 308, are deposited into the ALAT (4). The integration process begins when the next instance of *I11* is fetched and renamed. First, the input logical register is translated using the rename map (5a) while the PC is used to index the integration table for a potential match (5b). Since the translated and data-driven input registers match, *I11* is integrated by setting its output to the stolen physical register *p4* (5c). The rename map is updated accordingly (5d). Since *I11* is a load, integration is not complete. When its control-driven memory dependences resolve, *I11* probes the ALAT for a matching address. A match means that no conflicts have occurred and the *I11* does not need to re-access the cache (6). With *r0* mapped to *p4*, *I1* can now be integrated in a similar manner.

4 A Speculative Data-Driven Subsystem Implementation: A Speculative Dataflow Engine

The previous section described only the observed operation of the data-driven subsystem. We now describe a particular implementation, one that closely resembles a processor for the Tagged-Token Dataflow Architecture (TTDA) [3]. TTDA is an explicit dataflow architecture meaning it executes data-driven programs. TTDA is simple, elegant and

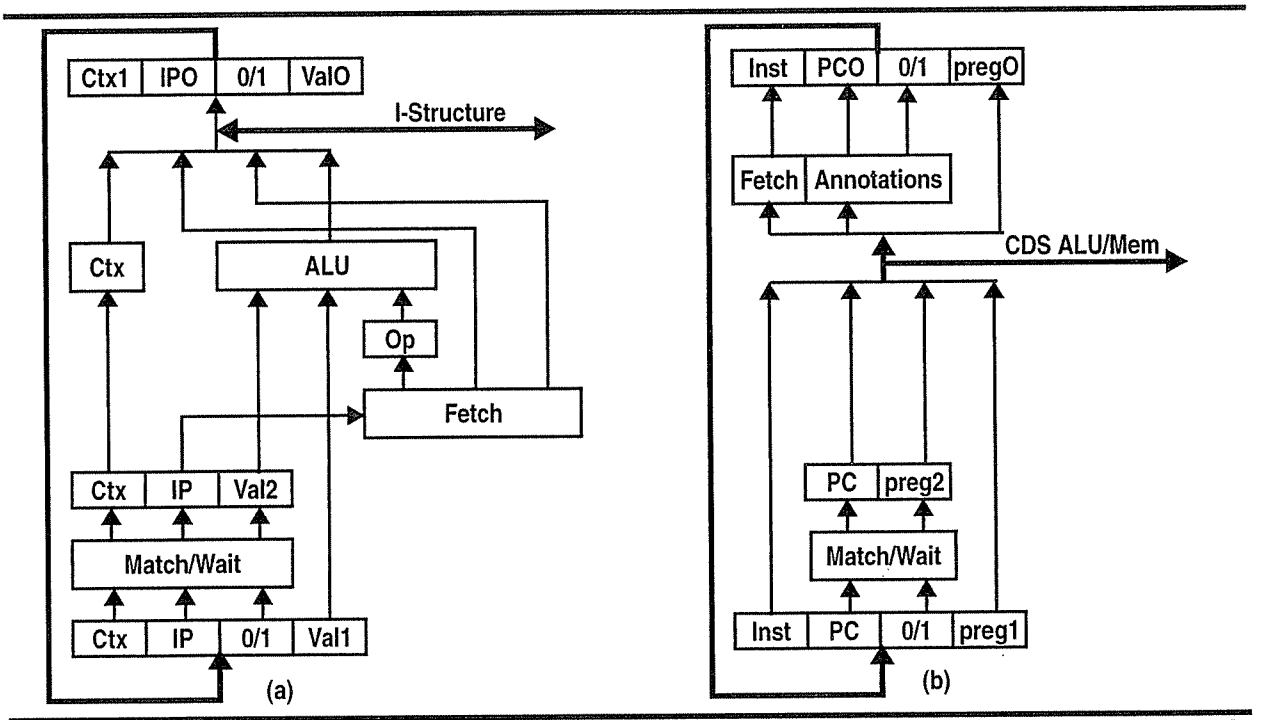

FIGURE 4. TTDA. (a) Original and (b) Modified to support speculative data-driven sequencing and execution. In the original TTDA, fetch precedes a synchronous ALU stage. The asynchronous interface is provided by the tail of the token queue. In the modified TTDA, fetch occurs after an asynchronous ALU stage.

manages resources at instruction granularity, making it good for both illustration and a first cut implementation. Although other implementations are possible, their complexity would serve only to lengthen the exposition.

4.1 TTDA: An Explicit Dataflow Architecture

The organization of a TTDA processor is shown in Figure 4(a). A TTDA processor is a worklist engine. The basic unit of work is the **token** and represents an instruction to be executed and one of its input operands. The worklist is called the **token queue**. Tokens are dequeued, executed in a **token pipeline**, and on completion trigger the generation of new tokens according to instruction specifications. Processing halts when the token queue is empty.

A TTDA executable is an explicit forward representation of the program's dataflow graph. A TTDA instruction consists of an opcode and up to two destination specifiers; the zero, one or two inputs are implicit. Each destination specifier is an **instruction pointer (IP)** and a **port bit** which specifies one of two inputs in the destination. Conditional execution (control) is represented via **switch** instructions which route one of the operands to one of the destinations based on the opcode and the value of the other operand.

A TTDA token is an **<instruction,value>** pair. In addition to IP:port information, the instruction specification includes a **context** tag that allows multiple dynamic instances of a static instruction to live in the machine simultaneously. A token's value portion stores a single temporary result, like a register in a von Neumann machine. Permanent value storage is provided in I-Structures, functional memories with per-address full/empty bits. The I-Structure controller stalls loads to empty addresses until a corresponding store arrives. A split transaction memory interface is

implemented elegantly by the enqueuing end of the token queue. Memory requests are sent to the memory system instead of the token queue and in turn the memory system responds to load requests by enqueuing tokens.

Token processing occurs in four stages. In the **match** stage, tokens for two-input instructions rendezvous with their second input by accessing a table in which other partially ready two-input instructions wait. Tokens for single-input instructions bypass the match step. Tokens with all their input operands proceed to **fetch**, where the IP portion is used to retrieve the corresponding instruction from an instruction store. During the **ALU/context** stage, operation execution and context manipulation for destination IP's proceed in parallel. Finally, in the **generate** stage, the computed value is concatenated with the fetched destinations specifiers to form and enquene zero, one or two new tokens.

4.2 A Modified TTDA for Speculative Data-Driven Execution

The SDD subsystem implementation we present is a dataflow engine that structurally resembles TTDA but, per interface requirements, internally manages physical register numbers and conventional memory instead of in-line values, context tags, and I-Structure requests. Physical register numbers replace both values and context tags, since multiple dynamic instances of the same static instruction can be assumed to have different associated physical registers. A pipeline diagram for a modified TTDA is shown in Figure 4(b).

In addition to the language difference, the modified TTDA pipeline has two structural differences. First, the asynchronous CD interface is not between the ALU stage and the token queue, it **is** the ALU stage itself. Second, instruction fetch and the retrieval of destinations for new tokens (these are logically separate functions in the modified version since data-driven sequencing directives are not part of imperative instructions) occur **after** the ALU stage. The queued tokens are expanded to hold the instruction itself in addition to its PC. There are several reasons for this organization. The data-driven subsystem shares physical registers with the control-driven subsystem implying that it also shares data paths and functional units. Asynchrony is built into the ALU stage to allow data-driven instructions to be scheduled on idle shared resources minimizing contention with the CD engine and allowing results to come back at arbitrary times. By placing the fetch/destination stage after ALU, we accomplish three things. First, we provide an easy path for computation initiation by making initiation requests look like completed SDD instructions. Second, we avoid toting destination information along the CD data paths. Finally, although we do expand the token by including the instruction in it, we do save a pipeline stage by implementing destination lookup and fetch together, potentially in the same table. This is actually an important point. For performance reasons, we do not want the data-driven execution engine to contend with the imperative engine for the instruction cache.

Since a TTDA activates instructions in a computation one at a time, the annotation structure required to drive a modified TTDA is also fine-grained and explicit in nature. The instruction granular representation simplifies initiation logic by allowing inputs to a computation to be supplied piecemeal as they become available in the control-driven subsystem. Our annotation table is a small cache indexed by PC that produces some small number of **<initiation bit, destination PC, port bit, instruction>** quads. The role of the destination PC and port bit has been explained. The presence of the actual instruction follows our choice for implementing fetch and data-driven destination prediction as

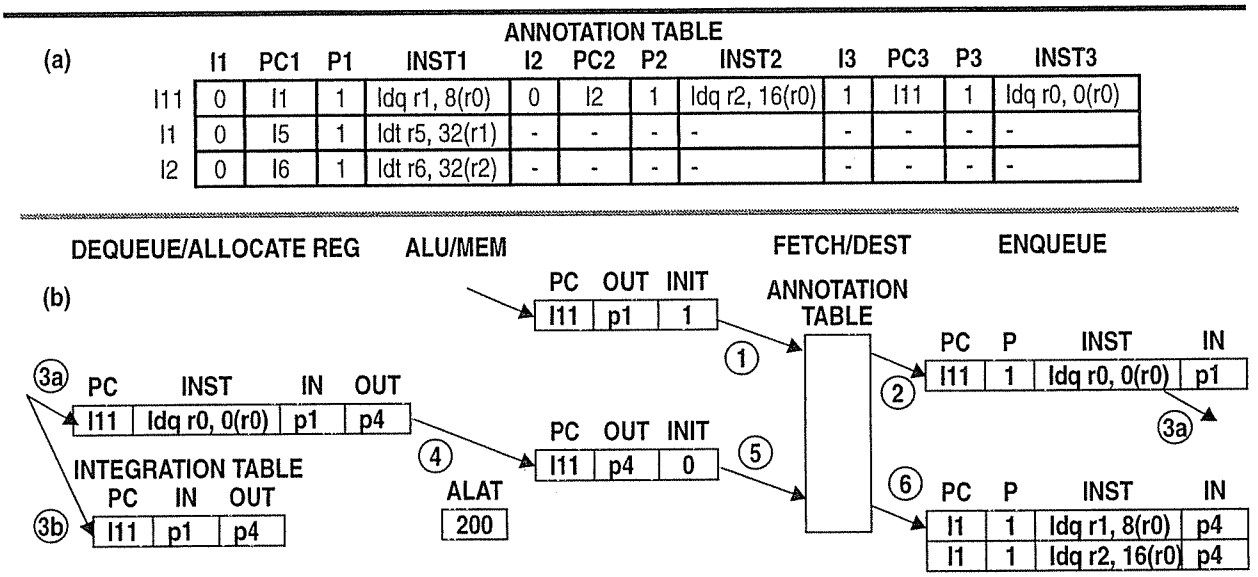

FIGURE 5. Working example: the data-driven side of initiation and execution. *(a) Annotation table representation of the miss-generating computation from Figure 1. (b) Sequencing through the first three instructions in the modified TTDA. Initiation occurs at the asynchronous ALU/MEM interface. The match stage is omitted since the code sequence contains no two input instructions that require rendezvous.*

a single pipeline stage. The initiation bit allows us to implement the selective initiation policy described earlier. Annotations for the miss-generating computation from Figure 1 are shown in Figure 5(a). Notice, the initialization bit is set only for the data-driven sequencing edge from *I11* to itself.

4.3 Initiation/Data-Driven Execution Example

Figure 3 showed the work involved in initiation and integration on the control-driven side. In that example, a triple *<I11, p1, 1>* was sent to the data-driven subsystem which responded by producing integration table and ALAT entries for the appropriate instructions. We illustrate the workings of that process in Figure 5(b). The *<I11, p1, 1>* triple enters the modified TTDA pipeline after the ALU/MEM stage. At the fetch/destination stage, it accesses the annotation table using the PC *I11* (action1), and retrieves the only entry with a matching initiation bit: *<I11:1:ldq r0, 0(r0)>*. This triple is concatenated with the register specifier *p1* to form and enqueue a token (2). This token is dequeued in the next cycle, bypasses matching and proceeds directly to the ALU/MEM stage where it is allocated a stolen physical register *p4*. The expanded token *<I11:ldq:p1:-:p4:0>* (**<PC:opcode:preg1:preg2:pregout:init>**) is then enqueued for execution on one the shared execution queues (3a). In parallel, a matching entry (minus the opcode and initialization bit) is created in the integration table (3b). An ALAT entry is created when the load actually issues (4). When the load completes, the triple *<I11:p4:0>* comes back along the initiation bus and heads for the annotation table (5). This time, entries matching *I11* with a clear initialization bit are extracted. Tokens *<I1:1:ldq r1, 8(r0):p4>* and *<I2:1:ldq r2, 16(r0):p4>* are created and enqueued (6). The process repeats until no further tokens can be generated.

5 Experimental Evaluation

Although our data-driven sequencing mechanism is fully general and can handle any computation, it is neither practical nor desirable to execute a significant portion of the program on it. In fact, in this preliminary evaluation we limit the fraction of dynamic instructions that execute in the data-driven subsystem to less than 10%. There are many reasons for choosing this division of labor. First, many instructions will not significantly benefit from data-driven execution. Integrating a data-driven instruction is not equivalent to removing it from the control-driven stream entirely. Since integration occurs during register renaming, integrated instructions must still be fetched and renamed, consuming valuable front end bandwidth. Unable to reduce front end latencies[1], data-driven pre-execution and subsequent integration is left to attack latencies caused only by **dependences**, **resource queuing** and **raw execution**. Of these, only load resource latency, load execution latency and non-load dependence latency are actually reduced (for non-load instructions, resource and execution latencies are typically negligible while load dependence-stalls are not hidden due to the effective re-execution requirement). Data-driven pre-execution of computations that do not exhibit these problems is not therefore useful.

Another reason for keeping the load on the data-driven subsystem low is a product of a limitation of our data-driven sequencing annotations. Specifically, the lack of TTDA style control in the representation makes pre-executing computations with complex control difficult. Our data-driven execution mechanism is adept at greedily executing an data-dependence graph, but cannot restrict execution based on control decisions. Limiting speculative data-driven execution to well defined computations with simple control reduces the amount of unnecessary work performed by that subsystem and destructive resource contention with the control-driven subsystem[2].

Our desire to minimize the load on the data-driven subsystem makes computation selection that much more important. Looking to maximize performance impact given a small static and dynamic portion of the program, we chose load latency due to L1 data cache misses as the target of our first experiment. This choice of problem does not emphasize the importance of the integration mechanism since most of the gain will be due to the prefetching effects of the data-driven subsystem. However, the problem is well documented, easily measurable, and since relatively few static instructions produce the majority of L1 cache misses [1], fits well both with our desire to restrict the dynamic load on the data-driven subsystem and our limited ability to hand select computations (explained next).

5.1 Methodology

Due to space limitations and the novelty of the mechanism, we restrict the focus of this evaluation to the intrinsic effectiveness of the technique. To that end, we attempt to factor out the problems associated with computation selection and representation. The representation of the computations chosen to execute in the SDD subsystem are loaded

1. Data-driven pre-execution can be used to attack front end latency indirectly, by expediting the resolution of mispredicted control decisions. However, that is not the focus of this paper.
2. Incorporating control information into our data-driven representation and sequencing framework is possible, but is outside the scope of this initial work.

in from a file at the beginning of the performance run into an "infinite" table, eliminating both cold-start and working-set effects. The annotated computations are selected semi-automatically from a previous run of the program using a different input data set. A first profiling pass selects static loads that contribute a high fraction of the L1 data cache misses in the program. A second pass records the dynamic computations (backward slices) that resulted in misses, analyzes them for statistical contribution and instruction distances and computes conservative guesses at potential "useful" computations. The computations are then examined manually. Ones with obviously bad timing characteristics are discarded. Others are pruned, shortened, or otherwise modified to improve expected behavior. Depending on the benchmark, anywhere from none to most of the computations may need some human intervention. For instance, *su2cor* has simple control and data flow and the computations generated by profiling are used as is. *Compress*, on the other hand has complex control and data flow and the automatically generated computations must be painstakingly pruned to achieve any performance.

The benchmarks we use are a collection of programs written in C and C++ collected from the SPEC95, Olden and OOCSB benchmark suites. The programs we selected showed non-negligible cache miss rates that were concentrated

Program	Description	Input Parameters: Ref (Train)
compress	LZ compression of randomly generated files	100000 q 2131 (1000 q 2131)
li	Lisp interpreter	boyer (8 queens)
su2cor	Monte-Carlo computation of particle masses	2 iterations (1 iteration)
em3d	3D electro-magnetic fields	4000 nodes, arity 10, 100 iters (8000, 20, 100)
mst	Dijkstra's minimum spanning tree algorithm	2048 nodes (1024)
deltablue	Integer constraint solver written in C++	30000 constraints (10000)

TABLE 1. Benchmark Programs.

Branch Prediction	8K entry combined 10-bit history gshare and 2-bit predictors. 2K entry, 4-way associative BTB, 32 entry return-address-stack.
Instruction Fetch	3-stage fetch. 32-entry instruction buffer. Up to 8 instructions from up to 4 basic blocks fetched per cycle.
Execution Core	8-way issue out-of-order core with a maximum of 128 instructions or 64 loads/stores in-flight. 256 physical registers are available. 2 cycle register read prevents fetch from Wrong path execution modeled. Loads and stores issue via a 64 entry queue with a 1 cycle load bypass. Loads wait for all previous store addresses before issuing.
Memory System	32KB, 32B lines, 2-way associative, 1 cycle access first level instruction cache. 64KB, 32B lines, 2-way associative, 1 cycle access, first level data cache. A maximum of 32 outstanding data misses. 16-entry ITLB, 32-entry DTLB with 30-cycle hardware miss handling. Shared 1MB, 64B line, 4-way, 12 cycle access second level cache. 70-cycle memory latency. 32B bus to L2 cache clocked at processor frequency. 16B bus to main memory clocked at 1/3 processor frequency. Cycle level bus utilization modeled.
Functional Units (latency)	8 int ALU (1), 3 int mult (3), 3 int div (20), 4 FP add (2), 3 FP mult (4), 3 FP div (24), 4 load/store (2) . The FP adders and all multipliers are fully pipelined.
Modified TTDA	3-stage TTDA pipeline with merged fetch/destination prediction stages. Up to 8 tokens generated per cycle. Infinite 8-way annotation table. 32-entry fully associative matching store. 32-entry token queue. 64-entry result buffer.

TABLE 2. Simulated Machine Configuration.

in fewer than 25 static loads, a cutoff value suggested by our only partially-automated computation selection process. The programs were compiled using the Alpha OSF `cc` and `c++` compilers with flags `-arch ev56 -O4`. Inspection of the compiled code showed that loop unrolling was used extensively. The programs were run to completion after a variable number of instructions were skipped to eliminate sometimes sizable initialization phases.

The simulation environment is built on top of the SimpleScalar 3.0 Alpha toolset. The cycle-level pipeline simulator has been modified to model register renaming, instruction replays, varying numbers of pipeline stages, write buffers, outstanding miss resources, and cycle accurate bus contention. It also simulates a modified TTDA. The details of the simulation environment are shown in Table 2. The modified hardware configuration was explicitly chosen to reduce destructive interaction between the CD and SDD subsystems. For instance, 256 physical registers were provided and these were logically partitioned to allow the results from data-driven computations to occupy a maximum of 64 at a time. Functional units were replicated so that stalls due to asymmetric resource availability are virtually eliminated. Support for a relatively large number of simultaneous outstanding memory requests is also included, allowing the data-driven subsystem to generate multiple cache misses (something it can do more rapidly than a control-driven processor) without stalling control-driven execution. Finally, the scheduler itself prioritizes ready work from the control-driven side over ready work from a data-driven computation. These measures effectively eliminate performance deg-

Metric	compress	li	su2cor	em3d	mst	deltablue
Instructions (M)	315.85	188.27	509.46	140.21	201.74	630.47
Loads (M)	67.17	48.84	113.52	56.61	48.69	172.50
Observed load L1 misses (M)	6.84	1.91	16.21	23.87	18.35	20.64
Observed load latency (cycles)	3.06	2.78	7.12	52.32	49.26	13.88
Static loads targeted	1	6	25	11	3	1
Static instructions annotated	11	18	88	19	7	10
Static dependences annotated	12	25	88	21	7	10
SDD insts executed (M)	21.51	12.57	27.62	17.61	14.59	28.87
SDD insts integrated (M)	17.71	3.78	27.48	13.25	14.44	28.87
SDD inst integration ratio	82.3%	30.1%	99.5%	75.2%	99.0%	100.0%
SDD inst load-factor	5.6%	2.0%	5.4%	9.5%	7.2%	4.6%
SDD loads executed (M)	3.93	8.64	13.22	16.09	10.42	19.12
SDD loads bypassed (M)	2.50	0.74	0	0	2.08	0
SDD loads integrated (M)	5.71	3.01	13.14	11.72	12.37	19.12
SDD loads mis-integrated (M)	0.00	0.18	0	0	0.15	0
SDD load integration ratio	88.8%	32.1%	99.4%	72.8%	99.0%	100.0%
SDD load load-factor	8.5%	6.2%	11.6%	20.7%	25.4%	11.1%
Observed load L1 misses (M)	3.94	1.19	12.46	15.35	13.66	4.11
Observed load latency (cycles)	2.98	2.53	6.29	29.15	42.66	11.42
Speedup	**4.1%**	**4.8%**	**4.9%**	**35.9%**	**13.7%**	**53.1%**

TABLE 3. Performance results. *All statistics dynamic unless stated otherwise.*

radation due to contention. Slowdowns are still possible, however, as there is no way to restrict the SDD subsystem's use of the L2 and memory busses.

5.2 Results

The results of our experiment are summarized in Table 3. The first four rows characterize the un-optimized run of the reference program including dynamic instruction and load counts and the number of L1 data cache misses and the average load latency observed for committed loads. The second group of rows is a static characterization of the computations selected to execute on the SDD subsystem. Computations are characterized by the number of static loads targeted and the total occupancy of the annotation table in terms of both instructions and dependences. A low ratio of annotated instructions to targeted loads implies that parts of computations are shared among many loads.

Dynamic performance data is summarized in three parts. The first part characterizes the overall behavior of the SDD subsyste. We measure the total number of instructions executed in the SDD subsystem and those that were subsequently integrated. The **integration ratio** is computed by dividing instructions integrated by instructions executed. The fact these ratios are high across the board testifies to the robustness of the integration mechanism and to the utility of the computations we selected. The **instruction load factor** characterizes the division of labor between the subsystems. It is obtained by dividing instructions integrated by program instructions committed. In all cases, the load factor is under 10% as we planned. The second part provides the same information but focuses on loads. We split SDD loads into executed and bypassed (not really executed) and also list the number of load mis-integrations. As we stated earlier, load mis-integration is an extremely rare event. Mis-integrations are caused when a data-driven load conflicts with a store that is not a part of its computation representation. Mis-integrations are rare because stores that cause conflicts with any regularity are simply incorporated into the computation. Load factor is higher for loads than for instructions because our computations specifically target loads. The first two sets of dynamic metrics indicate that speculative data-driven sequencing is doing its intended task. The last measures actual performance. Again, we measure the number of L1 data cache misses and the average latency observed for committed loads in the control-driven subsystem. Comparing miss counts can be misleading since partial misses (misses to previously requested blocks) are not counted. Comparing average load latencies paints a fuller picture. Relative miss counts are still interesting because they tell us how often the data-driven subsystem tried to overlap an actual would-be cache miss, whether or not it succeeded in fully doing so. Speedups are computed by dividing the cycle time of the base run by that of the optimized run and taking the difference from 1. A speedup of 100% implies that execution time was halved.

Performance-wise, our benchmark programs can be split into two camps. *Em3d*, *mst*, and *deltablue* have high miss rates, regular and relatively control-free computations for generating these misses and ample time between encountering the initiating instructions and the eventual cache misses to pre-execute the computations. This combination of ingredients is ideally suited for data-driven sequencing. The other benchmarks provide less ideal conditions. *Su2cor* has control-free computations and high cache miss rates, but also enough instruction level parallelism to hide most of these misses using conventional techniques. Compress has high miss rates, but tricky control flow and a tight inner

loop. Roughly 75% of the misses in *compress* cannot be hidden by data-driven sequencing due to either computations that are too control-complex or insufficient time in which to execute them. Most of the misses in *li* are embedded in conditionals leading to a lot of unnecessary work on the part of the data-driven subsystem unnecessary.

6 Related Work

Data-driven sequencing is an old idea. Explicit dataflow architectures which were developed in the 1970's and 80's [3, 6, 8, 14] replaced the von Neumann programming interface with a partially ordered data-driven representation. With all the ready-to-execute work in the entire program visible at any instant, explicit dataflow machines are able to exploit massive amounts of parallelism, and effectively tolerate extremely long latencies. However, the huge amount of available parallelism caused problems. Process state for a dataflow program is not of constant size, but proportional to the amount of parallelism. Consequently, dataflow machines could execute only programs with as much or less parallelism as processor state buffers [5]. In some cases, parallelism had to be artificially restricted to prevent resource deadlock, a task that proved difficult. Excessive fine-grain synchronization (incurred twice for every two-input instruction) was also a problem. A hybrid dataflow/von Neumann machine proposed to reduce this cost by restricting dataflow execution to coarse-grain synchronization and using control-driven execution between synchronization points [9]. Finally, the acceptance of dataflow architectures faced resistance from the software community as the partially ordered representation proved somewhat incompatible with the imperative programming languages that had become popular. Our work on speculative data-driven sequencing was inspired by explicit dataflow research. Speculative data-driven sequencing captures some of explicit dataflow's powerful capabilities: the ability to look beyond a control-driven instruction window to find parallelism and the ability to efficiently tolerate long latencies. Our technique uses speculation to provide all of these benefits micro-architecturally, making them available to imperative programs in a binary compatible fashion. By freeing the data-driven execution engine from program correctness responsibilities, we also allow these performance engines to be built in a simpler and more efficient manner.

Mechanisms for dynamically extracting data-driven representations of selected computations and performing controlled data-driven execution were developed initially for the purpose of prefetching linked data structures [15]. Data-driven pre-execution is the natural (really the only) transparent and scalable solution to a problem that has resisted more indirect forms of attack. The mechanism proposed in that work was a simple queue based scheme that did not execute two input instructions, did not explicitly share resources with a sequential engine and did not perform explicit integration (it was used for prefetching effects only). Later work by the same authors applied the mechanism to another problem, predicting the targets of virtual function calls [16]. That work included a primitive integration mechanism that ordered pre-computed targets and interfaced with the target predictor. However, intermediate results were not reused. Similar work aimed at accelerating branch resolution used the same high-level notions but a different execution model [7]. Speculative data-driven sequencing builds on this work, generalizes it, and introduces a simple but powerful integration scheme that allows work to be performed in a data-driven partially-ordered context and used directly in a control-driven totally-ordered context with little added effort and with certainty of correctness.

Recently, a short paper appeared describing a Data-threaded micro-architecture [20]. The data-threaded micro-architecture is similar to speculative data-driven sequencing, but far more aggressive. It proposes to dynamically convert significant chunks of the program to data-driven form and pack the annotations with enough control-flow and memory dependence information to allow instruction fetch and decoding to be discontinued and instruction retirement itself be data-driven. No evaluation of the micro-architecture or a discussion of its performance and practical implementation issues was given. Our proposal is more modest but, at this point, more realistic. First, we restrict the load on the data-driven subsystem to performance critical computations only. We also rely heavily on the control-driven subsystem to enforce architectural total ordering, give the illusion of sequential execution, and guarantee correctness. In the future, we may also want to increase the execution load on our data-driven partition, perhaps to as much as 50% of the program or more. Before doing so, however, we need to gain a much better understanding of the nature of this new execution model and its many delicate interactions.

Our general notion of using a supporting execution engine to accelerate a single control-driven thread has appeared in software-oriented forms as well. Assisted Execution [18] and Simultaneous Subordinate Microthreading (SSMT) [4] are both methods for software-controlled execution of selected functions for the purpose of pre-computing results or achieving beneficial side effects. Assisted execution relies on a simultaneous-multithreading processor and extremely lightweight thread creation mechanisms to implement software prefetching routines that are decoupled from the primary instruction window. SSMT programs micro-instructions into special on-chip storage and triggers their execution in software. In both cases, the supporting code is not a part of the main thread and cannot be reused directly.

7 Conclusions and Future Work

This paper introduces a technique for achieving data-driven sequencing for selected parts of unmodified imperative programs. This new technique is the centerpiece of a novel processor organization, CD/SDD, that allows a control-driven and a data-driven engine to cooperate in the execution of an imperative program. The power of speculative data-driven sequencing lies in its ability to isolate selected computations from their control-driven surroundings and execute them at their intrinsic rate without window and other control-driven sequencing constraints. By doing so, it greatly increases a processor's ability to find parallelism and aggressively overlap latencies. Our main contribution is the description of the organization and internal interface of the CD/SDD processor framework. Without restricting the implementation details of the data-driven subsystem, we present simple but powerful hardware algorithms for initiating data-driven execution of selected computations and subsequently integrating the results back into a control driven execution without having to repeat any of the work. The integration mechanism we introduce allows data-driven sequencing to play an increasingly active role in program execution. Due to the novelty of the framework and our relative inexperience with its subtleties, a full evaluation is not possible. However, a preliminary set of experiments focused at pre-executing computations that cause would-be data cache misses shows that even a modest implementation has the potential for great performance benefits on a variety of applications.

Perhaps the most exciting part of this work is the number of future research directions it opens. First, much work remains to be done to increase our understanding of the potential power of this new technique and where and how it may best be applied. Data cache misses are only one source of performance degradation in imperative programs. Control mis-speculations are another. Speculative data-driven sequencing and subsequent integration has great potential for providing seemingly instant perfect branch resolution (not perfect branch prediction, but rather perfect resolution at the rename stage). Other applications are certainly possible.

The most important part of the CD/SDD framework involves the selection of computations that are to execute in a data-driven manner and their representation. To limit the scope and length of the paper, we abstract this component away almost completely. Our experiments used profiler and human components for selecting computations semi-automatically. It is possible that a compiler may replace the human in this flow but, like any joint compiler/hardware technique, this approach raises questions about the communication interface. Virtual machine interfaces may work well here. Help from software may not be needed at all. There have been several detailed proposals of hardware mechanisms that can annotate computations dynamically. However, their ability to extract computations that include control more complex than loops is unproven. In the end, even speaking of requirements of mechanisms to automatically extract computations is premature as not much is known about the structural and statistical properties of performance-critical computations. A detailed study of these properties would be a good place to start.

Finally, the TTDA-style implementation we presented is simple and works nicely for the kind of loads we place on our data-driven subsystem. However, other implementations are possible. One promising alternative has the data-driven computations mapped onto threads in a simultaneous multithreaded [19] or speculative multithreaded processor. Placed in the context of our framework, such coarse-grain mapping would be an interesting micro-architectural analog of the hybrid data-/control-driven architecture referenced earlier.

8 References

[1] S. Abraham, R. Sugumar, D. Windheiser, B. Rau, and R. Gupta. Predictability of Load/Store Instruction Latencies. In *Proc. 26th International Symposium on Microarchitecture*, pages 139–152, Dec. 1993.

[2] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. In *Proc. 32nd International Symposium on Microarchitecture*, pages 226–236, Nov. 1998.

[3] Arvind and R. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, Mar. 1990.

[4] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous Subordinate Microthreading (SSMT). In *Proc. 26th International Symposium on Computer Architecture*, May 1999.

[5] D. Culler and Arvind. Resource Requirements of Dataflow Programs. In *Proc. 15th International Symposium on Computer Architecture*, pages 141–150, May 1988.

[6] J. Dennis and D. Misunas. A preliminary architecture for a basic dataflow processor. In *Proc. 2nd International Symposium on Computer Architecture*, pages 126–132, Jan. 1975.

[7] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes. In *Proc. 31st International Symposium on Microarchitecture*, pages 59–68, Dec. 1998.

[8] J. Gurd, C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, Jan. 1985.

[9] R. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proc. 15 International Symposium on Computer Architecture*, pages 131–140, May 1988.

[10] Intel Corporation. *Pentium Pro Family Developer's Manual*, 1996.

[11] Intel Corporation. *IA-64 Application Developer's Architecture Guide*, May 1999.

[12] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2), Mar./Apr. 1999.

[13] A. Moshovos and G. Sohi. Streamlining Inter-Operation Communication via Data Dependence Prediction. In *Proc. 30th International Symposium on Microarchitecture*, pages 235–245, Dec. 1997.

[14] G. Papadopoulos and D. Culler. Monsoon: An Explict Token-Store Architecture. In *Proc. 17th International Symposium on Computer Architecture*, pages 82–91, Jul. 1990.

[15] A. Roth, A. Moshovos, and G. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proc. 8th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, Oct. 1998.

[16] A. Roth, A. Moshovos, and G. Sohi. Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation. In *Proc. 1999 Internation Conference on Supercomputing*, pages 356–364, Jun. 1999.

[17] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *Proc. 22nd International Symposium on Computer Architecture*, pages 414–425, Jun. 1995.

[18] Y. Song and M. Dubois. Assisted Execution. Technical Report #CENG 98-25, Department of EE-Systems, University of Southern California, Oct. 1998.

[19] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. 23rd International Symposium on Computer Architecture*, pages 191–202, May 1996.

[20] D. Wilmot. Data Threaded Microarchitecture: A Radically Out-of-Order Microarchitecture for More Superscalar Performance. *Computer Architecture News*, 26(5):22–32, Dec. 1998.