# Speculative Data-Driven Multithreading

Amir Roth
Gurindar Sohi

Technical Report #1414

April 2000

# Speculative Data-Driven Multithreading

Amir Roth and Gurindar S. Sohi
Computer Sciences Department, University of Wisconsin - Madison
{amir, sohi}@cs.wisc.edu

## Abstract

*Speculative data-driven multithreading (DDMT) is a general purpose mechanism for expediting the execution of critical instructions like frequently mispredicted branches and loads that miss in the cache. A sequential processor has difficulties prioritizing these computations over other less performance-critical ones because it must fetch and rename all computations sequentially, regardless of their contribution to performance. To overcome this limitation, the computations of performance degrading instructions are annotated so that they can execute as standalone speculative threads. When a control-driven processor predicts an upcoming instance of a performance-degrading instruction, it forks the computation for that instruction as a data-driven thread. Control- and data-driven threads execute in parallel. However, since the data-driven thread fetches and executes only the critical computation, it can generate the critical result much faster than a control-driven thread which must fetch and execute all computations sequentially. By getting an early start on their computations, data-driven threads can consume the latencies of critical instructions before they have a chance to impact the externally-visible control-driven thread. Early results show that, when used to pre-execute the computations of would-be cache misses and branch mis-predictions, speculative data-driven multithreading can improve performance by up to 17%.*

## 1 Introduction

Program performance is measured by instruction retirement throughput. Since retirement is sequential, program performance is not uniform at the instruction level. Retirement typically proceeds at or near peak rate for several cycles, then stalls completely for some number of cycles when the oldest instruction in the machine cannot retire. The majority of retirement stalls occur for one of two reasons. Loads that miss in the cache stall retirement directly — they are unable to retire because they have long execution latencies. Mispredicted branches stall retirement indirectly — they may retire promptly but induce stalls in future instructions by delaying their processing. We call mispredicted branches and missing loads *critical* instructions.

Since critical instructions are responsible for the majority of lost throughput, it seems intuitive that processors would make their prompt execution a priority. However, in a sequential control-driven processor, even this conceptually simple goal is difficult to achieve. First of all, prioritizing only the critical instructions themselves is rarely sufficient. The computations of critical instructions, all instructions that transitively contribute values to the final result, must also be prioritized. At the same time, the number of high priority operations must be limited lest the entire program be prioritized and the process become counter-productive. While critical-computation-aware out-of-order scheduling might improve the situation slightly, a more significant problem is that instructions must be fetched and renamed before they can be scheduled. A sequential processor must fetch and decode all computations in order, regardless of their issue status and contribution to performance. Consequently, it often sequences low-priority, not-ready instructions before high-priority, ready-to-execute ones. Finally, compilers are likely to be of little help in these situations. Since critical computations are interleaved in the program with non-critical ones, the desired schedule will require algorithm-level code transformations, and, in the end, will likely shift criticality to new instructions.

In this paper, we present *speculative data-driven multithreading (DDMT)*: a general-purpose mechanism for eliminating or reducing stalls associated with critical instructions by allowing their computations to be sequenced directly while skipping over interleaved instructions from non-critical computations. The basic construct of this mechanism is the *data-driven thread*: a critical computation that speculatively executes in parallel with the rest of the sequential program. In this paper, we refer to the sequential program as the *control-driven thread.* Data-driven threads are composed of instructions from the original sequential program. However, since they need not be dynamically contiguous sequences, instructions belonging to data-driven threads must be annotated to enable data-driven sequencing and execution. Data-driven threads are speculative in two respects: their execution is speculative *and* the structure of the computations they encapsulate is also speculative. When a control-driven thread predicts an upcoming instance of a critical instruction, it "forks" the computation for that instruction as a data-driven thread. The data-driven thread speculatively executes in parallel with the control-driven thread. However, while the control-driven thread fetches and executes *all* computations *sequentially*, the data-driven thread fetches and executes only the critical computation. As a result, the data-driven thread often completes execution of the targeted critical instruction before the control-driven thread has even fetched it. In effect, the data-driven thread "absorbs" critical instruction latency before it can impact the control-driven thread.

One way a data-driven thread can assist a control-driven thread is by initiating would-be cache misses early. In general, however, the fact that a data-driven thread has completed a certain critical computation does not help the control-driven thread unless the latter can actually use the results of that computation. The second piece of DDMT is *integration*, a mechanism that performs a micro-architectural instruction-level "join" between a data-driven thread and a control-driven thread, passing pre-computed data-driven results to the control-driven thread as it register renames the corresponding control-driven instructions. Integration spares the control-driven thread from having to re-execute work performed in data-driven threads. In addition, by enforcing a one-to-one correspondence between control- and data- driven instructions, integration enables data-driven threads to be used for such applications as matching pre-computed branch outcomes with their intended dynamic branch instances.

Our implementation of DDMT is based on a simultaneous multithreading (SMT) [21] processor that can dedicate all its resources to a single thread of control when that thread can fully exploit them, or divide its resources among several threads. This model allows us to evaluate data-driven threads fairly by easily constructing systems with data-driven threads that have identical resources and bandwidths as purely control-driven systems. We evaluate DDMT as a technique for reducing the observed latencies of branch mis-predictions and cache misses. Our preliminary results show that, for both applications of the technique, good speedups are achievable over aggressive base configurations with limited additional hardware support.

The rest of the paper proceeds as follows. Section 2 provides a working example of DDMT and introduces some system components and their structure and function. The next two sections describe system aspects in more detail: Section 3 deals with the characterization, evaluation and automatic selection of data-driven threads; Section 4 expands on the hardware implementation focusing on the integration algorithm. Section 5 evaluates DDMT's effectiveness in

reducing resolution latencies of frequently mis-predicted branches and cache misses. Sections 6 and 7 discuss related and future work, respectively.

## 2 Working Example

We begin with a working example of data-driven multithreading. Through an example, we can be more concrete about the structure and operation of data-driven threads. An example lets us introduce the components of the system and their function at a high level. Subsequent sections will deal with individual components in greater depth.

Figure 1(a) shows (roughly) the dominant computation loop from the program *em3d*. The outer loop traverses a linked-list of nodes; the inner loop traverses an array of neighbor nodes performing computations on the outer node. The conditional statement in boldface contains a frequently mis-predicted branch. Figure 1(b) shows a sample execution of two outer-loop and three inner loop iterations in Alpha assembly. Instructions corresponding to neighbor computation, inner-loop control, and outer-loop control are shaded for clarity. The critical branches are instances of instruction *I8*. The narrow jags represent delay due to *I8* mis-predictions. We can reduce the penalty of these mispredictions by executing computations of *I8* as data-driven threads.

The precise data-driven thread we wish to execute consists of the boldface instructions in Figure 1(b). This thread comprises two *I8* computations from the first two inner-loop iterations of a single outer-loop. The first instruction in the thread is the outer loop induction, *I19*. Since the thread has no external input values other than the input to *I19*, it may be "forked" as soon as that input value is ready, namely at the *previous* instance of *I19*. We call an instruction that triggers a thread fork the *trigger instruction*. The trigger instruction typically supplies the last of the external inputs to the thread. It is clear that, starting from the trigger instruction *I19* forward, the control-driven thread has somewhat more instructions to fetch and execute than the data-driven thread before computing the outcomes the two *I8* instances. Thus, we expect the data-driven thread to improve the resolution latencies of these mis-predictions. Finally, notice that computations for instances of *I8* from the iteration immediately following the fork point are not included in the thread. We explain this and other decisions regarding thread selection in Section 3.

Now that we have decided what the data-driven thread should be, how do we go about using it? The first step is to make the processor aware that a data-driven thread exists. This is the job of the data-driven thread cache (DDTC). The DDTC contains a description of the thread in the form of an ordered list of its instructions' PCs. The PCs are necessary so that the control-driven thread can subsequently recognize the instructions; the full list is required because the dynamically dis-contiguous nature of the instructions prevents the representation from being compressed. In addition to the PCs, the instructions themselves may be represented inline in the DDTC to avoid serializing the sequence and fetch operations, and to reduce instruction cache contention. The DDTC is indexed by trigger instruction PC. The representation of our data-driven thread as it appears in the DDTC is shown in Figure 1(c).

Figure 1(d) illustrates the operation of a data-driven thread. Each of the important actions is denoted by a circled number. A control-driven thread can fork a data-driven thread when it decodes an instance of its trigger instruction

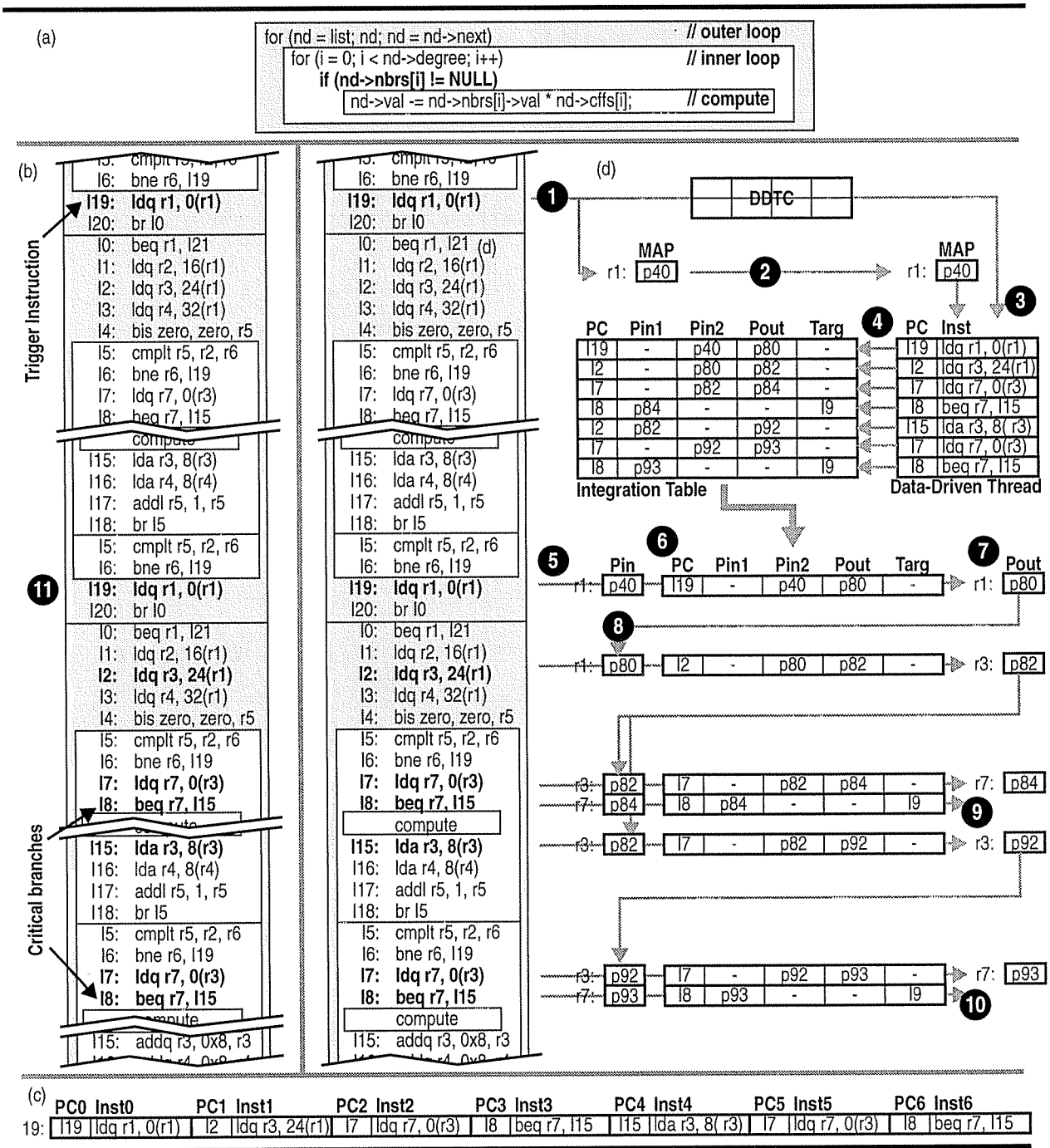


***FIGURE 1. Working Example.*** *(a) Loop code from EM3D with frequently mispredicted branch. (b) Dynamic control-driven execution of two outer loop iterations with computations of mispredicted branch instances in bold. (c) The data-driven thread representation of these computations in the DDTC. (d) Dynamic execution of same loop iterations augmented with data-driven thread that pre-computes the mispredicted branch outcomes.*

(Action 1). The data-driven thread is assigned to a free hardware context if one is available. The new hardware context is initialized with a copy of the control-driven thread's register rename map as it was at the time immediately following the renaming of the trigger instruction (Action 2); this process is similar to the fork operation of threaded multiple path execution (TME) [22]. Using its local copy of the register map, the data-driven thread fetches the computation and renames it in the usual way (Action 3), allocating new physical registers to new results while updating

the map to reflect these changes. The micro-fork map copy operation allows the data-driven thread a "natural" and architecturally precise way to both access results produced by the control-driven thread and to allocate storage for its own results without affecting control-driven state.

Armed with a copy of the register map, a hardware context sequences and executes a data-driven thread exactly as it would a control-driven instruction sequence, with three exceptions. First, since a data-driven thread is speculative it cannot affect architecturally visible state: stores in data-driven threads do not write to the cache and all faults are ignored. Second, an executing data-driven thread performs no explicit control-flow. The outcome of any control-flow instruction within a data-driven thread does not affect the execution path of the thread, it is simply saved for subsequent integration. Removing explicit control-flow simplifies implementation and prevents cyclic threads which can run unchecked. Although explicit control is not allowed, a data-driven thread may contain multiple instances of static instructions as well as instructions that dynamically occur along multiple and mutually exclusive paths. These concessions permit a data-driven thread to "simulate" some common control structures like loops by simply unrolling them. For example, the data-driven thread in our example contains two unrolled iterations of the inner-loop branch code. Third and finally, a data-driven thread can perform a special operation in which the input value of a store is forwarded to any number of loads with neither party computing an address. This operation is called a *data-driven cloak* and will be discussed further in Section 4.

The final component of DDMT is *integration*, in which individual instruction results computed in data-driven threads are seamlessly incorporated into the control-driven thread in what can be described as an instruction-level micro-architectural "join". Integration works by using physical register numbers to match up corresponding control-driven and data-driven dataflow graphs instruction by instruction. It exploits the fact that these graphs are rooted by the same physical registers, a guarantee made by the fork map copy operation. Integration is implemented as a part of control-driven register renaming: an instruction is integrated simply by setting the corresponding control-driven output architectural register to point to the corresponding data-driven output physical register. To enable integration, a data-driven thread records the instructions it has executed in the *integration table*, a structure that indexes the contents of the physical register file in a way that describes the dataflow graph of the computation (Action 4). Each integration table entry contains the instruction's PC, the physical register numbers of that instance's input and output and a target field for control instructions. In Figure 1(d), we can trace the dataflow graph of the computation in the integration table: *I19* has as its input *p40,* which was found via lookup in the data-driven thread's copy of the register map, and as its output *p80*, a newly allocated register; *I2* has as its input *p80*, the output of *I19,* and as its output *p82*, another newly allocated register, and so on.

During register renaming, the control-driven thread searches the integration table for a record of a data-driven instruction that corresponds to the control driven instruction it is currently examining. For instance, when it renames *I19* the control-driven thread sees that the input of that instruction is *p40* (Action 5). Looking in the integration table, it sees that there is an entry for an instruction *I19* with input *p40* and output *p80* (Action 6). At that moment, the control-driven thread knows that the instruction instance for which it is about to allocate a new physical register has actually

already been executed in a data-driven thread. Rather than allocate a new output register for the result, it simply sets the mapping for *r1* to point to the already computed value in *p80* (Action 7). Instruction *I19* is now integrated and need not re-execute if it has already executed in the data-driven thread! From here, the integration process continues recursively. When it reaches *I2*, the control-driven processor now sees *p80*, the register it has recently integrated, as its input (Action 8). Consulting the integration table, it finds an instance of *I2* with input *p80* and output *p82*. The input match results in the integration of *I2*, which in turn makes possible the integration of *I7* and so on and so forth. When the instances of *I8* are finally integrated, potential mis-predictions can be resolved immediately (Actions 9 and 10) reducing the penalty. The process repeats as the thread for the next *outer-loop* iteration is forked by the next instance of *I19* (Action 11). This arrangement overlaps control-driven execution of an outer-loop iteration with data-driven branch pre-computations for the next outer-loop iteration.

## 3 Data-Driven Thread Selection

Data-driven thread selection is the single most significant factor in determining the performance impact of DDMT. Good threads pre-fetch and pre-compute results that would have otherwise induced pipeline stalls and do so using minimal additional fetch and execution bandwidth. Bad threads pre-compute results that would not have caused stalls, do so no faster than a control-driven thread, and slow the latter down by consuming too much bandwidth. It seems appropriate that, of all aspects of DDMT, the problem of thread selection is the most difficult to characterize, the most slippery to formalize, and the most combinatorial to solve.

Our aim in this paper is to introduce the problem and some of the issues, describe some metrics for predicting and comparing the utility of data-driven thread candidates, and present a preliminary algorithm for creating data-driven thread annotations from program traces. Regarding this algorithm, our objective is to demonstrate that, insofar as it can be formalized, the process of selecting data-driven threads is automatable. In this paper, we assume an off-line implementation that communicates the annotations to the processor via the executable. In this scenario, the DDTC behaves like an instruction cache, loading threads from memory on a DDTC miss. DDTC miss detection may be implemented by annotating the trigger instructions in the normal text section. We believe that data-driven thread selection can be implemented in hardware. The feasibility of such an approach has already been demonstrated, albeit for less general threads [15, 16]. In such a scenario, the DDTC and any supporting structures would be purely micro-architectural, populated dynamically by thread-selection logic. However, a hardware implementation of the algorithm we present is outside the scope of this paper and is left for future work.

### 3.1 Evaluating the "Goodness" of Data-Driven Threads

Before we present our thread selection algorithm, let us characterize what a *good* data-driven thread is. Intuitively, a good data-driven thread is one that, when executed in parallel with its corresponding sequential region, will compute certain critical results faster than a control-driven thread executing the sequential region by itself. To make such judgements, we need to quantify *a priori* the difference in execution times between a given computation executing in a control-driven thread and that same computation executing as a data-driven thread.

| | | 4-wide machine | | | | | | 8-wide machine | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | control-driven | | | data-driven | | | control-driven | | | data-driven | | |
| | | I# | FC | FCDH | I# | FC | FCDH | I# | FC | FCDH | I# | FC | FCDH |
| I19 | ldq r1, 0(r1) | 23 | 7 | 8 | 1 | 1 | 2 | 23 | 3 | 4 | 1 | 1 | 2 |
| I2 | ldq r3, 24(r1) | 27 | 8 | 9 | 2 | 1 | 3 | 27 | 4 | 5 | 2 | 1 | 3 |
| I7 | ldq r7, 0(r3) | 32 | 8 | 10 | 3 | 2 | 4 | 32 | 4 | 6 | 3 | 1 | 4 |
| I8 | beq r7, I15 | 33 | 9 | **11** | 4 | 2 | **5** | 33 | 5 | **7** | 4 | 1 | **5** |
| I15 | lda r3, 8(r3) | 40 | 10 | 11 | 5 | 3 | 4 | 40 | 5 | 6 | 5 | 2 | 4 |
| I7 | ldq r7, 0(r3) | 46 | 12 | 13 | 6 | 3 | 5 | 46 | 6 | 7 | 6 | 2 | 5 |
| I8 | beq r7, I15 | 47 | 13 | **14** | 7 | 4 | **6** | 47 | 6 | **8** | 7 | 2 | **6** |

***TABLE 1. Computing fetch-constrained dataflow heights (FCDH).*** *I# is the distance in dynamic instructions from the trigger, sequential for data-driven threads but not necessarily so for control driven threads. FC is the fetch constraint, obtained by dividing I# by the available fetch width.*

We estimate execution times using the *fetch-constrained dataflow-height* (FCDH), a composite metric that recognizes the importance of both data-dependences and limited fetch bandwidth. For conventional dataflow calculations, the input dataflow height (DH) of an instruction represents the time at which the instruction becomes data-ready and is computed as the maximum of the output dataflow heights of those instructions on which it is data dependent. The output dataflow height represents the time at which the instruction is complete and is computed by adding the latency of the instruction to its input dataflow height. For FCDH, we modify the calculation of the input height at every instruction to include a fetch constraint (FC). Thus, the input FCDH represents the earliest time at which the instruction is *both* data-ready *and* fetched. The fetch constraint itself can be calculated in several ways; a simple one divides the instruction's dynamic distance from the trigger instruction by the available fetch width. Since instruction distances from the trigger are always lower in a data-driven context, it is this fetch term that promises that a given computation will execute at least as fast in a data-driven thread as it would in a control-driven thread.

In Table 1, we revisit our running example from Section 2 and compute the FCDH for the two branches in the data-driven thread we selected. For simplicity, we assume that all operations have unit latencies. There are four separate calculations in the table, each represented by three columns. The first calculation is for a control-driven thread with a fetch width of 4. The first column (I#) is the instruction number: its distance the trigger instruction. The fetch constraint (FC) is computed from the instruction number by dividing by the fetch width (4) and rounding up. The fetch-constrained dataflow-height (FCDH) for each instruction is computed using the dataflow rule and the fetch constraint. For example, *I15* has a fetch-constraint of 10 cycles and its input, *r3,* is produced by *I2* which has an output height of 9. Since 10 is greater than 9, the fetch constraint is active and the height of the instruction is 10 plus the instruction latency, or 11. Proceeding this way, we compute the FCDH for the two branches as 11 and 14. In the second group of bars, we repeat the computation for a data-driven thread. However, to simulate the data-driven thread sharing fetch bandwidth with a control-driven thread, we compute with a fetch width of 2 instead of 4. Here, since the instruction numbers are sequential, the fetch constraints are lower and we calculate the data-driven FCDH as 5 and 6. The calculations tell us that on a 4-wide issue machine the data-driven thread we selected should provide the critical branch outcomes 6 and 8 cycles earlier than they would have been available otherwise. Repeating the calculations using an

8-wide machine, we find that the advantage shrinks to 2 cycles per branch. This should not surprise, since wider fetch is essentially a brute-force attack on the same sequencing limitations that DDMT overcomes using selectivity.

The FCDH is a good metric in that it encapsulates our intuition about data-driven threads. However, it is not perfect. FCDH is simultaneously optimistic and pessimistic. It is pessimistic in that it oversimplifies the control-driven fetch model, ignoring intermediate branch mispredictions and other sources of fetch under-utilization. It also ignores the nature of the other work in the control-driven thread. It is optimistic in that it abstracts away the notion of total system performance, comparing control-driven execution to data-driven execution rather than to contentious simultaneous execution. Unfortunately, to be of any use, composite calculations not only need to know the nature of the other work in the control driven thread, but require analytic detail that is almost equivalent to full simulation.

Finally, we can address the question: "why was the first post-trigger instance of branch *I8* not included in the data-driven thread?" Simply, its control-driven distance from the trigger was not sufficiently long for a data-driven thread to provide a result faster than a control-driven thread. At the same time, adding the extra computation to the data-driven thread would have reduced the data-driven fetch advantage for the other two branches. All the same, the excluded instance of *I8* can be pre-computed by a data-driven thread triggered by the previous instance of *I19*. This example also gives some flavor of the combinatorial difficulties associated with data-driven thread selection, where the benefit of including a given computation in a data-driven thread must be weighed not only against its resource impact on the control-driven thread but also against other data-driven computations!

### 3.2 An Algorithm for Extracting Threads from a Program Trace

We now present an algorithm for extracting data-driven threads from program traces. As we mentioned earlier, although an offline implementation is assumed, the logic is the main point. Input to the thread construction and selection algorithm is an annotated program trace. The annotations mark which load instances missed in the cache, which branch instances were mis-predicted and the addresses of all loads and stores. Both trace and annotations can be generated on-the-fly using functional simulation. Data-driven threads are constructed by walking the trace backwards starting from a misbehaving instance of a critical instruction. Only the 1K instructions immediately preceding the critical instance are considered. Empirically, growing the trace window beyond 1K instructions does not significantly change the traces produced. The intuitive reason for this is that the bulk of a computation for a given result happens close to that result [23]. As it walks backwards through the window, the thread constructor will add an instruction to the thread if it (a) satisfies an active register or memory dependence for an already included instruction or (b) is itself an annotated critical instruction. When an instruction is added, the dependence it satisfies is removed from consideration and its own input dependences are added. When a load is added to the thread, on-line address information is used to decide on the spot whether to pursue its address slice or a slice through a possible cloaking store. Note that the thread contains no explicit control: the dynamic path is implicit in the instruction list.

Instructions are added to the thread until one of two termination conditions is reached. First, if the total number of instructions in the thread exceeds a certain limit, thread construction ends with failure on the dual arguments that (a)

the data-driven thread will fetch too many instructions and excessively slow down the control-driven thread and (b) the encapsulated dataflow graph is already too large to achieve a practical fetch advantage. Second, whenever the distance from the last instruction added passes a certain threshold, thread construction transitions into trigger-finding mode and is guaranteed to terminate successfully. The reasoning here is that a single large distance at the beginning will provide adequate fetch advantage for the entire thread. That such large distances exist is both intuitive and supported by empirical evidence. In our running example, most of the leverage is supplied by the outer loop induction, *I19*. To give thread selection the best chance to succeed, the thread constructor repeats the selection process for each dynamic critical instruction multiple times, each time allowing fewer critical instructions to be added along the way. It is this feature that, in the running example, facilitated the discovery that including only two branch instances gave a better data-driven thread than including three. This process continues even after a thread with a satisfactory distance from the trigger to the first instruction has been found, as the thread selector attempts to maximize the FCDH difference between a control-driven and data-driven execution.

Once the program trace is completely processed, the extracted threads are pruned based on an inclusion hierarchy and relative frequencies. Thread selection can be tuned using thread-size, minimum trigger distance, trace window size and control- and data- driven fetch width parameters as well as a flag that controls whether all instances or only misbehaving instances of critical branches and loads are candidates to start data-driven threads.

## 4 Hardware Implementation Aspects

We implement DDMT on top of a simultaneous multithreading (SMT) processor [21]. An SMT processor allows the entire width of the machine to be dedicated to a single thread if that thread has sufficient ILP. Multithreading support is included for those times when a single thread cannot fully utilize the machine. Compaq's Alpha 21464 [7] is such a processor. An SMT processor is a good environment for implementing and evaluating data-driven threads. From an implementation standpoint, it already supports out-of-order issue logic and register renaming, required for data-driven instruction scheduling and integration, respectively, while the flexible resource allocation policy allows data-driven threads to be implemented cheaply, "stealing" whatever bandwidth the control-driven thread is unable to exploit. From an evaluation angle, an opportunistic SMT allows us to easily construct purely control-driven and hybrid control-/data- driven system with identical resources. Several modifications must be made to the basic micro-architecture for DDMT to function properly. These are shown schematically in Figure 2(a) and include a DDTC, integration logic and support for cloaking. We expand on these and other implementation issues in this section.

### 4.1 Cloaking

As mentioned earlier, data-driven execution uses a speculative operation called *cloaking* in which a store forwards its input value to the output value of a load with neither of the participants ever computing an address. Cloaking is useful in a purely control-driven setting [13] but its importance is magnified in DDMT. The presence of cloaking enables the construction of data-driven threads that exploit the often long dependence arcs of predictably communicating store-load pairs to achieve fetch leverage while avoiding the overhead of fetching and executing the address calcula-

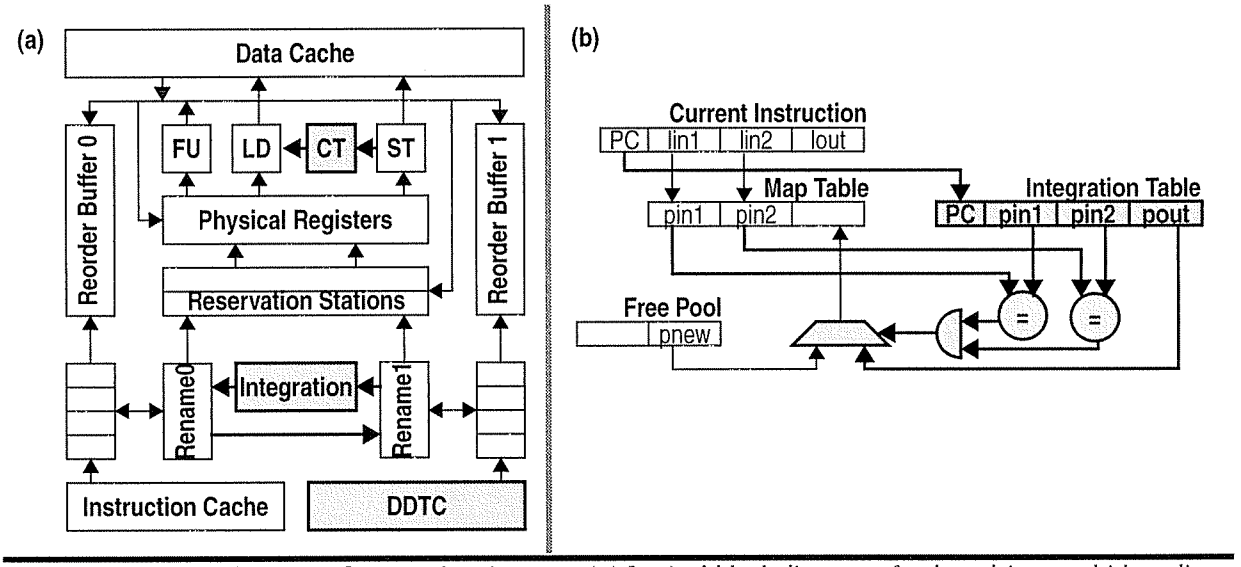


***FIGURE 2. Hardware Implementation Aspects.*** *(a) Logical block diagram of a data-driven multithreading enabled SMT processor. In an actual implementation, the rename tables, instruction buffers and re-order buffers would be shared. They are distributed for illustration purposes. DDTC is the data-driven thread cache. CT is the cloaking table. (b) Logical block diagram of integration. Unshaded logic is used in conventional register renaming. Shaded logic is integration-specific.*

tion. The most frequent example of this is a thread that contains a register save-restore pair across a sizeable procedure body.

To implement cloaking, data-driven loads are annotated with the PC of the store that supplies their input value. Data-driven loads with empty cloaking annotations are executed using address generation and cache access. Stores that participate in cloaking are also annotated with a single bit. In hardware, cloaking uses the *cloaking table (CT)*, a small buffer indexed by store PC. A store deposits a value in the entry corresponding to its own PC; a load picks up a value from the entry corresponding to its annotated store's PC. Similar to basic cloaking, the CT provides primitive synchronization to allow loads to wait for their corresponding stores and stores to wake up waiting loads [13].

## 4.2 Integration

The integration mechanism allows results from data-driven instructions to be re-incorporated directly into the control-driven thread without the need for re-execution. Integration allows a data-driven thread to perform entire computations on behalf of a control-driven thread. Without it, work performed by a data-driven thread can benefit a control-driven thread only indirectly, for instance by prefetching data. Integration exploits the facts that both control- and data-driven threads build dataflow graphs of their computations in the physical register file and that the data-driven fork register-map copy operation supplies a common set of physical registers to "root" corresponding computations. It then recursively matches up control- and data-driven dataflow graphs instruction by instruction using physical register numbers to represent the dataflow relationships. This process is enabled by the *integration table*, a new structure which describes the dataflow graphs of data-driven threads in terms of physical registers.

Figure 2(b) shows the integration operation in logical block form. The unshaded structures and thin paths correspond to conventional register renaming structures and logic. The shaded structures and thicker paths correspond to the additional logic required for integration. Conventional renaming consists of two logical steps: first the logical input registers of a instruction are mapped to physical register numbers using map table lookups, then the output logical register is allocated a free physical register and the mapping is recorded in the map table. Integration adds the following logic. The instruction PC is used to probe the integration table in parallel with the map table lookup. When both lookups complete, the corresponding physical register numbers are compared pair-wise. The output of the comparison is used to select the physical register that will be assigned to the output of the renamed instruction and entered into the map table. A successful comparison chooses the register from the integration table. If integration fails, the newly allocated register is chosen. Like plain register renaming, register renaming with integration is an inherently sequential process that permits a parallel prefix formulation to allow multiple simultaneous operations. Integration may also be pipelined alongside register renaming.

One implementation concern is the use of a set-associative integration table capable of buffering multiple data-driven instances of a single static instruction. Such an organization improves integration rates, but increases the number of physical register tag comparisons required exponentially. A solution for this problem is outside the scope of this paper; possibilities include managing the associative ways as a queue rather than a set or allowing fewer integrations (not renamings) per cycle.

One final implementation note concerns the physical register semantics of data-driven and integrated instructions. The semantics of when physical registers are freed by these instructions differ from those of conventional control-driven instructions. The differences are straightforward and lead to simple guarantees that physical registers will not be "leaked". We do not discuss this aspect further due to space requirements.

### 4.3 Load Re-execution

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. 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 thread to know about these stores since it has no notion of computations outside the ones it is charged with executing, much less the ordering between them and its loads. Integrating loads using physical register criteria only (addresses are not available at register rename time) jeopardizes correctness by permitting *mis-integrations*, integrations of possibly incorrect values. Empirically, load mis-integrations are extremely rare events. 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 a control-driven thread. 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. Since mis-integrations are rare, a cheap recovery solu-

tion like squashing and re-fetching all instructions downstream from the offending load can be used. At first, it may seem that mandating re-execution means that every integrated data-driven load accesses the data cache (or store queue) twice -- once in data-driven mode and once in control-driven mode. However, this situation can be made the exception rather than the rule. First, consider cloaked data-driven loads. In data-driven mode, these don't execute in the traditional sense; store to load forwarding occurs without cache access, address disambiguation or even address calculation. Cloaked loads actually execute only once -- in control-driven mode. In fact, re-execution makes speculative cloaking possible by acting as a verification step. However, even conventional loads that calculate addresses and access the cache can avoid re-execution. This is done using a technique that is the micro-architectural analog of architectural load speculation mechanisms like the Memory Conflict Buffer (MCB) [9] and IA64's Advanced Load Address Table (ALAT) [12]. The mechanism uses a small cache of addresses. Data-driven loads deposit their addresses in this structure while control-driven stores (from both within the processor as well as from other processors) remove matching addresses from it. Once an integrated load is disambiguated in a control-driven thread, it simply checks this cache for a matching address. A match means that no conflicting store occurred since the time the load was initially executed. Hence, the load need not access the data cache again. The address-buffer mechanism and cloaking mesh nicely with each other to ensure that every eventually integrated load actually executes at least once and that the majority execute at most once.

# 5 Performance Evaluation

We now quantify the performance potential of DDMT. We evaluate the technique in two roles: reducing the observed latency of would-be first-level data cache misses and reducing the sequential penalty of mis-predicted branches. The baseline system we compare to is an 8-wide SMT similar in spirit, but perhaps not in internal organization, to the Alpha 21464 [7]. Our DDMT system uses additional hardware in the form of a DDTC, and integration and cloaking logic. The sizes of these structures are small and we assume that they have no impact on cycle time or number of pipeline stages.

## 5.1 Methodology

The benchmarks we use are a collection of programs selected from the SPEC95, SPEC2000 and Olden benchmark suites. Due to space constraints, we select six programs for each experiment, two from each benchmark suite. Programs are selected for their (relatively) high branch mis-prediction or data cache miss rates, respectively. The programs are compiled for the Alpha EV6 architecture by the Digital UNIX V4 `cc` compiler with optimizations `-O3 -fast`. All programs are simulated in their entirety. The data-driven thread selection algorithm permits a maximum thread length of 32 instructions, requires a trigger distance of 64 dynamic instructions and considers only mis-behaving instances as potential thread seeds. FCDH metrics are computed using control- and data- driven fetch widths of 8 and 4, respectively. For each experiment, the data-driven thread selection phase uses a smaller (and if possible different) input data set than the DDMT performance measurement phase.

| | |
|---|---|
| Branch Prediction | 16K 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 12 instructions from two cache blocks fetched per cycle with a maximum of one taken branch per cycle. |
| Issue Mechanism | 8-way out-of-order speculative issue with a maximum of 128 instructions or 64 loads or 32 stores in-flight. 256 physical registers. 2 cycle register read. Loads speculatively issue in the presence of earlier stores with unknown addresses. The load and subsequent instructions are squashed and re-fetched on a mis-speculation. Store to load queue bypass takes 1 cycle. Loads and branches have the highest issue priority. Priority within a group is determined by age. |
| 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 16 outstanding load misses. 16-entry store buffer. 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. |
| Threads | 4 hardware contexts that share the instruction queue in round-robin fashion. No thread-prioritization beyond the fetch stage. |
| Data-Driven Thread Support | 16-entry, 1-cycle access data-driven thread cache (DDTC) with a maximum of 32 instructions per thread. A 128 entry 4-way set associative integration table. 16-entry cloaking table. |

***TABLE 2. Simulated machine configuration.***

The simulation environment is built on top of the SimpleScalar 3.0 [3] Alpha toolkit. The cycle-level simulator models an SMT with nominal stages: fetch, register-rename, queue, schedule, execute, writeback and commit but a variable number of pipeline stages for each function. We model a memory system with non-blocking caches, finite write-buffers and miss-status holding registers (MSHR), and cycle accurate bus-utilization. We simulate 4 hardware contexts which share all bandwidth, buffers, and execution resources. Our DDMT configuration, therefore, executes a single control-driven thread and up to three concurrent data-driven threads. Thread priority is made explicit in the fetch stage only, where bandwidth is allocated in round-robin fashion among all active threads on a cycle basis. Table 2 shows the simulation parameters in detail.

## 5.2 An Experiment Targeting First-Level Cache Misses

Our first experiment uses DDMT to target the latencies of loads that miss in the first level data-cache. Although a significant portion of second level cache hit latency can be hidden by a machine with 128 instruction lookahead and reordering capability, DDMT's absence of sequential window constraints allow it to further increase memory-level parallelism (MLP). Results for six benchmarks are summarized in Table 3. All the quantities represented are dynamic and counted in millions of events. No program uses more than 11 distinct data-driven threads, enough to fit into a 16-entry DDTC without replacement. Performance improvements range a negligible 0.1% speedup for *li* to a 17.3% speedup for *mst*. The other metrics help explain these numbers.

We measure load latency as the average difference between the issue and completion times of every *committed* load. L1 MSHR occupancy measures MLP. In most of the cases, average load latency decreases while MLP increases suggesting that the data-driven threads are overlapping misses that are further away than a single control-driven instruction window's worth. The exception is *mst*, where the dominant data-driven thread encapsulates a hash table search including hash function calculation. Execution of this thread overlaps a second linked list traversal with the one tak-

ing place in the control-driven thread, doubling the MLP but increasing memory bus contention to a level that negatively impacts the average load latency.

The observed speedups are somewhat smaller than the latency and MLP diagnostics suggest. The reason for this is that data-driven threads contend for resources with the control-driven thread, slowing it down. We approximate the full effect of resource contention by comparing the number of instructions fetched by each system. Two fetch counts we are particularly interested in are the number of wrong-path instructions fetched by the control-driven thread and the number of instructions fetched by data-driven threads. One noticeable trend is that DDMT reduces the number of instructions fetched along the wrong path. There are two effects here. First, a lower sequentially observed load latency indirectly reduces branch mis-prediction resolution time. The dominant effect, however, is contention due to data-driven threads. When overhead and contention increase to the point of offsetting all benefit, as is the case for *li*, DDMT should be suppressed. However, a mechanism for doing so dynamically is beyond the scope of this paper.

One troubling statistic is the number of fetched data-driven instructions that are eventually integrated. The table lists integration counts broken down by the status of the instruction at the time it was integrated. Instructions integrated in the completed and issued states have done useful work on behalf of the control-driven thread. Instructions integrated in the dispatched state are useless, because they did not save the control-driven thread any work. The best thing we can say about these instructions is that they indicate that the data-driven thread was trying to do the right thing, just

| | | | **compress** | **li** | **gzip** | **vpr** | **em3d** | **mst** |
|---|---|---|---|---|---|---|---|---|
| Base | Insns committed (M) | | 331.39 | 1188.37 | 3367.27 | 692.50 | 248.88 | 230.77 |
| | Insns fetched (M) | Total | 807.48 | 1890.49 | 5883.09 | 1352.76 | 379.34 | 232.62 |
| | | Wrong-path | 476.09 | 700.88 | 2493.11 | 660.05 | 130.46 | 1.85 |
| | Loads (M) | | 42.68 | 302.63 | 677.78 | 198.37 | 71.59 | 32.58 |
| | L1 misses (M) | | 3.08 | 3.46 | 23.12 | 8.46 | 24.50 | 4.16 |
| | Avg. load latency (cycles) | | 3.63 | 2.70 | 2.76 | 3.41 | 42.41 | 19.85 |
| | Avg. MSHR occupancy | | 1.89 | 1.51 | 0.83 | 1.67 | 10.76 | 0.94 |
| DDMT | Threads forked (M) | | 3.57 | 4.32 | 26.28 | 7.09 | 0.80 | 0.52 |
| | Insns fetched (M) | Total | 886.29 | 1905.92 | 5884.64 | 1474.34 | 403.64 | 248.95 |
| | | Wrong-path | 450.81 | 685.80 | 1794.18 | 635.97 | 131.56 | 1.80 |
| | | Data-driven | 104.08 | 29.20 | 671.02 | 145.24 | 23.20 | 16.24 |
| | Insns integrated (M) | Total | 16.39 | 16.19 | 269.20 | 39.76 | 20.83 | 14.34 |
| | | Completed | 15.82 | 13.38 | 248.52 | 37.14 | 12.82 | 10.30 |
| | | Issued | 0.20 | 1.51 | 6.58 | 0.90 | 3.90 | 1.13 |
| | Critical loads integrated (M) | Total | 2.36 | 5.08 | 22.43 | 6.18 | 11.24 | 4.82 |
| | | Completed | 2.12 | 4.00 | 13.80 | 5.04 | 5.61 | 2.52 |
| | | Issued | 0.15 | 0.86 | 5.45 | 0.69 | 2.91 | 0.69 |
| | Re-executed loads (M) | Cloaked | 5.79 | 2.97 | 71.61 | 10.76 | 0.00 | 0.52 |
| | | Address | 0.00 | 0.03 | 0.00 | 0.39 | 0.00 | 0.00 |
| | Load latency (cycles) | | 3.34 | 2.44 | 2.11 | 3.37 | 29.58 | 21.19 |
| | Avg. MSHR occupancy | | 3.05 | 2.50 | 1.05 | 1.81 | 12.58 | 1.81 |
| | Speedup over base | | **1.6%** | **0.1%** | **15.4%** | **12.0%** | **7.2%** | **17.3%** |

***TABLE 3. Using data-driven multithreading to pre-execute loads that miss in the L1 cache.***

didn't have sufficient time in which to do it. The integration factors we observe are lower than 60% for all benchmarks and as low as 16% for *compress*, and the useful integration factors are lower still, although most integrated instructions have performed some work already. This tells us that a significant number of data-driven threads are forked unnecessarily (at least part of the thread is not needed). This is due in part to working set differences in data sets used for thread selection and thread application, but also to inherent limitations in the threads themselves. Specifically, absent control, a data-driven thread will execute all computations within it, even ones that are not control-equivalent with the trigger instruction. Adding control-flow to data driven threads, allowing useless threads to terminate early may overcome this problem, but is outside the scope of this work. To complete our integration statistics, numbers are provided to substantiate our earlier claims about the utility of cloaking and the rarity of non-cloaked load re-execution.

### 5.3 An Experiment Targeting Branch Mis-predictions

Our second experiment uses data-driven threads to pre-compute the outcomes of frequently mis-predicted branches. This application is particularly attractive because, in addition to accelerating branch resolution and the fetch of correct-path instructions, it reduces the total number of instructions fetched and alleviates some of the fetch pressure created by DDMT. Results for six benchmarks are shown in Table 4. Aside from speedup, an important diagnostic metric is the branch mis-prediction resolution latency which we calculate as the average number of cycles between the mis-prediction of a branch and its completion. For the DDMT system we also measure the number of mis-predicted branches resolved by integrated data-driven instructions. A mis-prediction resolved by a completed integrated instruction is resolved instantly, for these DDMT accomplished its mission. Integrations of non-completed branches may still improve performance if most of the *computation* of the branch has completed in data-driven mode.

| | | | compress | ijpeg | gzip | vpr | em3d | bh |
|---|---|---|---|---|---|---|---|---|
| Base | Insns committed (M) | | 331.39 | 519.36 | 3367.27 | 692.50 | 248.88 | 977.44 |
| | Insns fetched (M) | Total | 807.48 | 816.16 | 5883.09 | 1352.76 | 379.34 | 2084.65 |
| | | Wrong-path | 476.09 | 296.80 | 2493.11 | 660.05 | 130.46 | 1098.09 |
| | Branches (M) | | 42.68 | 43.69 | 351.44 | 76.68 | 22.80 | 96.64 |
| | Mis-predictions (M) | | 3.06 | 3.71 | 15.83 | 4.77 | 1.30 | 11.21 |
| | Resolution latency (cycles) | | 40.50 | 12.49 | 33.34 | 50.04 | 74.05 | 25.84 |
| DDMT | Threads forked (M) | | 3.71 | 1.23 | 29.27 | 8.15 | 1.12 | 12.41 |
| | Insns fetched (M) | Total | 863.64 | 766.60 | 5793.42 | 1426.01 | 393.01 | 2008.65 |
| | | Wrong-path | 425.36 | 215.72 | 1764.48 | 530.83 | 126.85 | 843.23 |
| | | Data-driven | 106.90 | 31.52 | 565.41 | 202.47 | 17.28 | 178.80 |
| | Insns integrated (M) | Total | 25.46 | 18.95 | 312.58 | 46.58 | 9.03 | 52.47 |
| | | Completed | 24.69 | 15.98 | 273.06 | 42.36 | 7.84 | 52.16 |
| | | Issued | 0.04 | 0.87 | 5.71 | 0.18 | 0.33 | 0.16 |
| | Mis-predictions integrated (M) | Total | 0.35 | 1.59 | 4.79 | 0.75 | 0.07 | 4.60 |
| | | Completed | 0.35 | 1.31 | 2.60 | 0.74 | 0.07 | 4.56 |
| | Resolution latency (cycles) | | 36.95 | 9.53 | 21.39 | 43.41 | 59.60 | 22.38 |
| | Speedup | | **2.1%** | **7.4%** | **12.9%** | **5.3%** | **6.2%** | **2.9%** |

***TABLE 4. Using data-driven threads to pre-compute outcomes of branches that are likely to be mis-predicted.***

For the branch pre-computation application, we obtain speedups in the range of 2.1% for *compress* to 12.9% for *gzip*. The speedups are in proportion with the reductions in mis-prediction resolution latency. Also, in each case we see that the number of instructions fetched along the wrong path is decreased, although for some programs (*compress*, *vpr*, *em3d*) that number is more than offset by the additional fetch bandwidth consumed by the data-driven threads. This fetch (and by extrapolation execution) overhead of data-driven threads does not seem to entirely correlate with performance, suggesting that these overheads can be absorbed at least in part. Resolution latency reduction, which reflects the utility and timeliness of the data-driven threads appears to be more important.

## 5.4 Contributions of Data-Driven Sequencing and Integration

In our introduction we claimed that data-driven sequencing, namely the ability to *sequence* through critical computations faster than a window-constrained control-driven thread, is an important performance enabling aspect of data-driven threads; simply prioritizing critical computations while using conventional sequencing is insufficient. We also maintained that integration is an important component of the system, especially for order-sensitive applications like early branch resolution. In this section, we quantitatively verify these claims by attributing the performance of DDMT to each factor. The results are reported in Table 5. Due to space considerations, we choose a total of six benchmarks, three from each DDMT application.

To measure the importance of data-driven sequencing, we simulate the program using data-driven thread annotations as prioritization hints to the scheduler rather than as templates for data-driven threads. As we suspected, critical-computation priority-scheduling in hardware is largely ineffective, and often harmful. Prioritizing parts of critical computations that are already in the instruction window does not increase parallelism, it only delays execution of the oldest instructions in the machine, preventing them from retiring and freeing up slots for future potentially critical computa-

| | | Cache Misses | | | Branch Mis-predictions | | |
|---|---|---|---|---|---|---|---|
| | | **li** | **vpr** | **mst** | **ijpeg** | **gzip** | **em3d** |
| Base | Instructions fetched(M) | 1890.49 | 1352.76 | 232.61 | 816.15 | 5883.09 | 379.34 |
| | Load latency | 2.70 | 3.41 | 19.85 | 2.19 | 2.76 | 42.41 |
| | Resolution latency | 14.18 | 50.04 | 291.63 | 12.48 | 33.34 | 74.05 |
| DDMT | Instructions fetched (M) | 1905.92 | 1474.34 | 248.95 | 766.60 | 5793.42 | 393.01 |
| | Load latency | 2.44 | 3.37 | 21.19 | 2.19 | 2.34 | 42.07 |
| | Resolution latency | 13.86 | 41.60 | 207.36 | 9.53 | 21.39 | 59.60 |
| | Speedup over base | **0.1%** | **12.0%** | **17.3%** | **7.4%** | **12.9%** | **6.2%** |
| Base with "critical scheduling" | Instructions fetched (M) | 1892.94 | 1381.67 | 232.63 | 838.05 | 5900.50 | 381.34 |
| | Load latency | 2.68 | 2.84 | 19.85 | 2.19 | 2.76 | 42.12 |
| | Resolution latency | 14.42 | 50.92 | 291.70 | 13.40 | 33.10 | 73.37 |
| | Speedup over base | **-1.1%** | **-1.0%** | **-0.1%** | **-2.5%** | **0.4%** | **0.3%** |
| DDMT without Integration | Instructions fetched (M) | 1905.27 | 1499.82 | 248.81 | 844.65 | 6456.13 | 394.38 |
| | Load latency | 2.55 | 3.38 | 12.18 | 2.19 | 2.54 | 41.94 |
| | Resolution latency | 14.05 | 42.96 | 203.38 | 12.56 | 27.12 | 59.62 |
| | Speedup over base | **-1.2%** | **10.6%** | **22.2%** | **-1.0%** | **1.4%** | **%6.1** |

***TABLE 5. Measuring the contributions of data-driven sequencing and integration.***

tions. The power of DDMT comes from its ability to expose parallelism by using data-driven sequencing to increase the effective scheduling window.

To quantify the contribution of integration, we simply repeat our experiments but don't integrate the work performed in the data-driven threads. Intuitively, integration plays a bigger role in the pre-execution of branches than in the pre-execution of loads. Most of the impact of pre-executing a cache miss comes from the pre-fetching effect, integration provides only modest additional benefits. Integration is more critical for fast branch resolution because every cycle added to the mis-prediction penalty delays the processing of future correct path instructions. Counter-intuitively, the addition of integration actually *slows down* some prefetching applications. *Mst*, in particular, runs 5% faster without integration than with it. How is this possible? The reason is rather subtle. Instructions can be integrated in any stage of execution. Instructions that are integrated before being issued in the data-driven thread don't improve performance, in fact, they hurt performance because they occupy slots in the reservation stations, re-order buffer and load and store queues. Now imagine that integration is disabled. When the first instruction in a data-driven thread is not integrated, the rest of the thread becomes un-integrable, detached from the control-driven thread by virtue of its seed physical register becoming "stale". This stale mapping triggers a cascade of invalidations that results in the freeing of all slots occupied by instructions in the corresponding data-driven thread (essentially, integration increases overhead by allowing useless data-driven instructions to hold on to resources longer). In exchange for this overhead reduction, integration for completed instructions is forfeited. As we have seen, however, for DDMT targeted at cache misses, integration is less important since pre-fetching provides most of the benefit. Conversely, for the branch resolution DDMT application, integration is much more important and its exclusion is unlikely to produce speedups.

## 6 Related Work

Long latency loads and mis-predicted branches degrade performance in large part because they and their computations are bound into a sequential order that sometimes prevents their execution from proceeding at its maximum rate. DDMT "frees" critical computations from these sequential constraints allowing them to execute as fast as their data dependences permit. Starting in the 1970's, this approach to parallelism formed the basis for explicit dataflow architectures [5, 10, 11, 2, 14]. With the structure of computations exposed at the architectural interface, dataflow machines can approach theoretical levels of parallelism and latency tolerance. However, their acceptance has been slowed by engineering problems and an unpopular programming model. Speculative data-driven multithreading attempts to bring some of the performance aspects of these machines into the realm of sequential programs, albeit on a smaller and more task-specific scale.

The idea of using speculative threads to accelerate sequential programs is also not new. The multiscalar architecture [18], single-program speculative multithreading (SPSM) [6], thread-level data-speculation (TLDS) [20], and dynamic multithreading (DMT) [1] partition a program into *control-driven speculative threads*. Threads are partitioned either statically (multiscalar, TLDS, SPSM) or dynamically (DMT), forked in software (TLDS, SPSM) or hardware (multiscalar, DMT) and initiated in oldest-first (multiscalar), youngest-first (DMT) or compiler-controlled (TLDS, SPSM)

order. Speculative results are incorporated as control of the architectural state passes from the most recently committed thread to the least speculative remaining thread. Multiscalar, SPSM, DMT and TLDS are mechanically similar to DDMT but diametrically opposed in their philosophy. Control-driven speculative multithreading exploits primarily thread-level parallelism. For these techniques to succeed, the program must be partitioned into threads that are parallel or at least nearly parallel. As a group, control-driven speculative threads must have high degrees of control-equivalence, data-independence and load-balance. In contrast, DDMT primarily exploits instruction-level parallelism. Data-driven threads perform implicit control speculation, follow data-dependences closely, and are automatically load balanced by an opportunistic SMT. Data-driven threads do not require thread-level parallelism in the control-driven sense to be effective. They have the flexibility to attack individual latencies and points of performance loss and generally operate at high speculative efficiency levels. Of course, the price for flexibility and efficiency is that instructions in data-driven threads are not spared from being fetched and renamed in a control-driven thread. As we have seen, this overhead can limit the applicability of data-driven threads unless care is taken.

Assisted execution [19] and SSMT [4] are techniques for accelerating a sequential program by using idle threads to prefetch data, pre-set the branch predictor or execute other performance enhancing tasks. The code that runs on these threads, however, is not an annotated part of the original program that can later be integrated. Instead, it is a supporting piece of code (assisted execution) or microcode (SSMT) generated by the compiler or by hand and managed explicitly. Assisting and SSMT threads are auxiliary in a true sense; they can impact performance only indirectly and cannot do any real work on behalf of the main thread.

Dependence based prefetching and branch pre-execution [15, 16] and the branch-flow micro-architecture [8] are closest in spirit to DDMT. Each of these techniques extracts true data-driven threads from a sequential program, executes them and attempts to re-use the results to one degree or another. Dependence-based prefetching uses post-commit dependence-detection machinery to build a specialized representation of pieces of the dataflow graph. Given a seed value, this representation is traversed/executed on the side by a finite-state-machine. Integration is not performed. Dependence-based branch pre-execution builds on this mechanism and adds an ad-hoc correspondence mechanism to allow pre-computed branch results to be matched with dynamic branch predictions. The branch-flow micro-architecture is a different implementation of the same idea.

Dynamic instruction reuse (IR) [17] is mechanically similar to integration. However, IR forfeits some power and implementation simplicity in the realm of speculative result recovery in order to exploit general (non-speculative) instruction repetition. Integration is tailored towards recovering the results of a speculative execution.

## 7 Summary and Future Work

Data-driven multithreading (DDMT) is a novel speculation model. Speculation is performed at the granularity of a *data-driven thread*, a sequence of potentially non-consecutive dynamic instructions that represents a computation. At the point in a control-driven program at which the inputs to the data-driven thread become available, the data-driven thread is micro-architecturally forked and executed in parallel with the sequential program. However, while the con-

trol-driven thread sequentially fetches and executes the entire program, the data-driven thread deals with only those instructions that constitute the computation of interest. Consequently, it typically reaches and executes these computations much faster than would be possible for a sequential thread. DDMT can improve the performance of sequential programs when computations of loads that are likely to miss in the cache and branches that are likely to be mis-predicted are chosen as data-driven threads. By pre-executing these critical instructions, data-driven threads can absorb latencies that would otherwise impact sequential performance.

One particularly interesting component of DDMT is integration, a facility for incorporating results produced in data-driven mode into a control-driven thread while avoiding the need for re-execution. Integration works by using data-flow relationships expressed as physical register numbers to prove that a data-driven instruction and a control-driven instruction correspond to the same dynamic instance. Integration enables data-driven threads to perform actual work on behalf of a control-driven thread, including order-sensitive tasks like the pre-computation of branch targets.

One attractive substrate for implementing DDMT is an SMT processor. This kind of processor already contains register renaming and the flexible resource allocation policies needed to run data-driven threads efficiently. Furthermore, the additional hardware required to support data-driven threads is not prohibitive. Integration is the only logic that must be added to the processor's critical timing path, and its circuit complexity should not be much different than that of superscalar register renaming. Aside from integration support, a small cache placed in parallel with the instruction cache is required to house the threads. Support for cloaking is also needed, although such functionality may be available anyway due to its benefits for sequential execution.

Our experiments show that DDMT can be used effectively to reduce performance degradations due to cache misses and branch mis-predictions. In the latter case it improves performance while also *reducing* the total number of instructions fetched and executed by the machine. Overall, data-driven pre-execution shows promise as a unified general-purpose performance engine, able to attack any source of latency without the use of any problem-specific micro-architectural gadgets and structures.

There are several avenues for future work in this area as well. One problem we have noticed is the moderately frequent execution of a data-driven thread when its computation, or at least the critical instruction at the end of it, is not eventually executed. Our simplifying choice to exclude explicit control flow from data-driven threads is responsible. The addition of control-flow to data-driven threads will reduce the overheads of such useless threads by allowing them to exit early. It also has the potential to create self-perpetuating threads which may be useful for attacking very long latencies. A preliminary study of the contribution of control-flow to the computations of critical instructions has been conducted [23]; the results suggest that inserting useful control-flow that does not in itself exact excessive overhead is challenging. Our choice of an opportunistic SMT as a base micro-architecture was made for reasons of both implementation simplicity and pending availability. However, architectures and micro-architectures that support speculative control-driven threads are on the horizon. Mapping DDMT onto these architectures is certainly possible and the interaction between control- and data- driven speculative multithreading is interesting. Finally, the thread

selection algorithm we present shows promising success, but it is preliminary. Certainly, an implementation of our or any other algorithm in hardware would ease the acceptance of DDMT by moving the technique into the purely micro-architectural realm. We are experimenting with such an implementation.

# 8 References


[1] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. In *Proc. 31st International Symposium on Microarchitecture*, pages 226–236, Nov. 1998.

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

[3] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, Jun. 1997.

[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] 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.

[6] P. Dubey, K. O'brien, K. O'brien, and C. Barton. Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading. In *Proc. 1995 Conference on Parallel Architectures and Compilation Techniques*, pages 109–121, Jun. 1995.

[7] J. Emer. Simultaneous Multithreading: Multiplying Alpha's Performance. Microprocessor Forum, Oct. 1999.

[8] 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.

[9] D. Gallagher, W. Chen, S. Mahlke, J. Gyllenhaal, and W. Hwu. Dynamic Memory Disambiguation Using the Memory Conflict Buffer. In *Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, Oct. 1994.

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

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

[12] Intel Corporation. *IA-64 Application Developer's Architecture Guide*, May 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] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. In *Proc. 24th International Symposium on Computer Architecture*, pages 194–205, Jun 1997.

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

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

[20] J. Steffan and T. Mowry. The Potential for Using Thread Level Data-Speculation to Facilitate Automatic Parallelization. In *Proc. 4th International Symposium on High Performance Computer Architecture*, pages –, Feb. 1998.

[21] 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.

[22] S. Wallace, B. Calder, and D. Tullsen. Threaded Multiple Path Execution. In *Proc. 25th International Symposium on Computer Architecture*, pages 238–249, Jun. 1998.

[23] C. Zilles and G. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proc. 27th International Symposium on Computer Architecture*, Jun. 2000. To appear.