Speculative Data-Driven Multithreading

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

Abstract

Mispredicted branches and loads that miss in the cache cause the majority of retirement stalls experienced by sequential processors; we call these critical instructions. Despite their importance, a sequential processor has difficulty prioritizing critical computations (computations of critical instructions), because it must fetch all computations sequentially, regardless of their contribution to performance. Speculative data-driven multithreading (DDMT) is a general-purpose mechanism for overcoming this limitation.

In DDMT, critical computations are annotated so that they can execute standalone. When the processor predicts an upcoming instance of a critical instruction, it microarchitecturally forks a copy of its computation as a new kind of speculative thread: a data-driven thread (DDT). The DDT executes in parallel with the main program thread, but typically generates the critical result much faster since it fetches and executes only the critical computation and not the whole program. A DDT "pre-executes" a critical computation and effectively "consumes" its latency on behalf of the main thread. A DDMT component called integration incorporates results computed in DDTs directly into the main thread, sparing it from having to repeat the work.

We simulate an implementation of DDMT on top of a simultaneous multithreading (SMT) processor and use program profiles to create DDTs and annotate them into the executable. Our experiments show that DDMT pre-execution of critical loads and branches can improve performance significantly.

1 Introduction

Sequential program performance is measured by instruction retirement throughput. Performance degrades when retirement stalls waiting for the oldest active instruction to complete. The majority of retirement stalls are due to loads that miss in the cache and, indirectly, to mispredicted branches which delay the fetch and completion of future instructions. We call misbehaving branches and loads *critical* instructions.

Since they are responsible for the majority of lost throughput, it seems intuitive that processors would prioritize critical instructions. However, in a *control-driven* (or *sequential*) processor, this conceptually simple goal is difficult to achieve. First, prioritizing only the critical instructions themselves is rarely sufficient — the *computations* of the critical instructions, the instructions that transitively contribute values to the critical instruction, must also be prioritized. At the same time, the number of high-priority operations must be limited lest the entire program be prioritized. A more significant problem is that instructions must be *sequenced* (i.e., fetched and renamed) before they can be scheduled, and a sequential processor must sequence instructions in program order, regardless of their contribution to performance. Using a compiler to isolate critical computations from their surroundings at the program level is also difficult. It often requires algorithm-level code transformations, and may shift criticality to new instructions.

Speculative data-driven multithreading (DDMT) is a general-purpose mechanism that expedites the execution of critical computations by allowing them to be sequenced directly while skipping over dynamically interleaved instructions from non-critical computations. DDMT exploits the fact that only a few static critical computations produce the majority of dynamic critical instruction instances. These computations are annotated so that they can execute standalone. When the *main* thread (which we also call the *sequential* or *control-driven* thread) predicts an upcoming instance of a critical instruction, it "forks" a copy of its computation as a new kind of speculative thread — a *data-driven thread (DDT)*. A DDT is data-driven in the sense that it is not necessarily a dynamically contiguous instruction sequence. The main thread continues fetching and executing the entire program sequentially. In parallel, the DDT fetches and executes only the critical computation often completing execution of the critical instruction before the main thread has even fetched it. A DDT "absorbs" a critical computation's latency on behalf of the main thread.

DDTs execute *speculatively*, they do not change the architected state of the machine. As such, it would appear that the only way a DDT can assist the main thread is indirectly — for instance by initiating would-be cache misses early. However, DDMT can take advantage of a technique called *register integration* [18] to allow the main thread to directly use results computed in DDTs. Integration exploits the fact that both main thread and DDT place results in a shared physical register file. Using a modification to register renaming, integration allows the main thread to recognize and claim DDT results. Integration spares the main thread from having to re-execute (but not re-fetch) DDT instructions. It also enforces a one-to-one correspondence between DDT and main thread instructions, a property we use to match pre-computed branch outcomes with their intended dynamic branch instances.

In this paper, we present an implementation of DDMT that is based on simultaneous multithreading (SMT) and an algorithm for creating DDT annotations from program traces. Our experiments using this framework show that using DDMT to pre-compute frequently misbehaving branches and loads improves performance over aggressive base configurations.

The next section presents an overview and working example of DDMT. Sections 3 and 4 describe two DDMT aspects, DDT selection and hardware implementation, in more detail. Section 5 evaluates DDMT's effectiveness in reducing the impact of mispredicted branches and cache misses. Sections 6 and 7 discuss related and future work.

0-7695-1019-1/01 $10.00 © 2001 IEEE

2 Working Example

As an overview of DDMT, we begin with a working example. The top of Figure 1 shows a simple computation loop that traverses a linked list of nodes. Computation for each node checks for the existence of and then accesses a neighbor node. This is a simplified version of a computation from *em3d* in which each node loops over an array of neighbors. The neighbor-test branch and neighbor data-access in bold are two critical instructions. The bottom of Figure 1 shows a sample execution of parts of four loop iterations. The instructions of a single iteration are shaded to match the source code. Critical branch (*I3*) and load (*I5*) instances are in darker shade and, although it is not explicitly represented, we wish to "absorb" their latencies by pre-executing their computations as DDTs.

The DDT we choose consists of the boxed instructions in the trace — *I10, I10, I2, I3* and *I5*. The first boxed instance of *I10* (marker 1) is actually *not in the DDT itself,* rather it is the *DDT trigger* — the instruction *after* which the DDT is forked. The trigger is typically chosen to be the instruction that computes the *last* external input to the DDT. In our example, the trigger *I10* provides the only register input to the DDT — *r1* which contains node. The DDT executes two instances of the loop induction instruction *I10* (node = node->next) (markers 2 and 3), effectively jumping two iterations ahead of the main thread, then executes *I2, I3* and *I5*, the computations that lead from the induction variable to the branch and load.

Our choice of DDT (and trigger) is meant to maximize the execution advantage, with respect to the critical computation, the DDT has over the main thread. The main thread and DDT execute in parallel starting from the trigger. To get to the critical computation (marker 3) from this point, the main thread must fetch 22 instructions while the DDT must only fetch 2. Notice, *I3* and *I5* computations from the two iterations *immediately following* the trigger (markers 4) are *excluded* from the DDT. Their proximity to the trigger means that a DDT has little if any advantage over the main thread in sequencing and executing them. This certainly does not mean that any I3 or I5 instances are ignored. Each iteration's critical computation is forked by the induction instruction of two iterations before. Notice, it *is* possible for a single dynamic instruction (marker 3) to be logically part of multiple DDTs (the ones launched from markers 1 and 2). Integration ensures that DDTs physically share such instances.

Inside the processor, DDTs are *statically* represented by the *data-driven thread cache* (*DDTC*) (marker 5) — a structure resembling a trace-cache [15]. The DDTC is indexed by DDT trigger PC and represents DDTs as ordered lists of instructions. Each DDTC instruction is tagged with its PC because PCs are needed to recognize DDT instruction results during integration and because DDT instructions are not sequential.

The main thread forks the DDT when it decodes an instance of its trigger instruction, *I10* (marker 1). A new hardware context is allocated to execute the DDT and initialized with a copy of the main thread's post-trigger rename map (marker 6). This *rename-map fork* provides the DDT with a natural and architecturally precise way to access results produced by the main thread (via map translations), to allocate storage for its own results without affecting main-thread state (by allocating physical registers and changing only its private mappings), and to communicate results among its own instructions (again via map translations). A similar operation is used (for similar reasons) in threaded multipath execution (TME) [28]. It also provides a common mapping for starting the integration process.

DDTs are fetched and executed like conventional (control-driven) traces — performing *no explicit control flow.* DDT instructions are not sequential, so control flow has little meaning for them. The outcomes of DDT control instructions (like *I3*) are saved for later integration but do not affect subsequent DDT instructions. In lieu of explicit control, a DDT may implicitly represent any control flow (ours contains parts of two "unrolled" loop iterations). This arrangement simplifies implementation and prevents runaway cyclic threads, while still letting DDTs perform perfectly general computations.

A DDT executes speculatively — its results are not made architecturally visible. However, all DDT instruction results are entered into the physical register file and indexed in an *integration table (IT)* (marker 7). The IT indexes physical registers by their dataflow properties — the instruction (PC) and physical registers used to compute the stored value. The IT also contains a target field that remembers the pre-computed

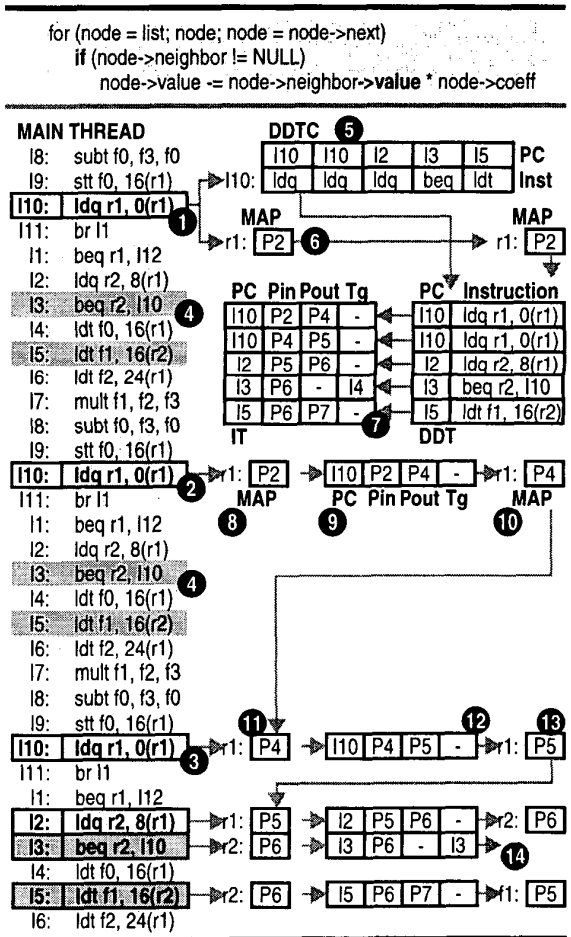


FIGURE 1. Working Example. (Top) Loop with frequently mispredicted branch and frequently missing load. (Bottom) Execution of parts of four iterations augmented with a DDT that pre-executes the branch and load computation. The computation results are subsequently integrated.

targets of control instructions. The IT in the figure contains a description of the dataflow graph of our DDT. Physical register *P4* holds the result of *I10* which was executed with physical register *P2* (found via lookup in the DDT's copy of the map) as input. *P5* holds the result of the second instance of *I10* which was computed with *P4* as input, and so on.

When the main thread itself starts renaming the critical computation, it uses the IT to locate the results of the corresponding DDT instructions. For instance, when it renames *I10* the main thread sees that the input of that instruction is *P2* (marker 8). Looking in the IT, the main thread sees that *P4* holds a value created by instruction *I10* with input *P2* (marker 9). The main thread recognizes P4 as having been created by an instruction that corresponds to the one it is currently renaming. It "claims" this result by mapping the output of the current instruction to *P4* (marker 10). Its own *I10* is now *integrated* and need not re-execute if it has already executed in the DDT. Integration proceeds recursively. When it renames the second instance of *I10*, the main thread sees the recently integrated *P4* as its input (marker 11). In the IT, it finds an instance of *I10* with input *P4* and output *P5* (marker 12). The input match results in the integration of *P5* (marker 13), which in turn enables the integration of I2, and so on. When *I3* is integrated, its mis-prediction can be resolved immediately (marker 14).

3 Data-Driven Thread Selection

Data-driven thread selection is a significant factor in determining the performance impact of DDMT. Good DDTs pre-compute results that would have otherwise induced pipeline stalls using minimal additional fetch and execution bandwidth. Bad DDTs pre-compute results that would not have caused stalls, do so no faster than a control-driven thread, and slow the system down by consuming too much bandwidth. Our aim in this paper is to introduce the problem of DDT selection, describe a new metric for estimating the utility of DDT candidates, and present an algorithm that uses this metric to create DDT annotations from program traces.

In this paper, we assume a profile-driven off-line implementation that communicates the annotations to the processor via the executable. The same approach is taken by the Multiscalar architecture [9, 22]. A hardware implementation of this algorithm which would make DDMT entirely microarchitectural may also be possible. Simple hardware has been proposed for extracting restricted threads [16, 17] and mechanisms for general threads are currently being investigated.

3.1 Measuring Utility of Data-Driven Threads

Intuitively, a good DDT is one that, when executed in parallel with its corresponding sequential region, will execute the critical computation faster than a control-driven thread executing the sequential region by itself. To identify good DDTs, we need to quantitatively estimate *a priori* the difference in execution times between a given computation executing in a control-driven thread and that same computation executing as a DDT.

We estimate execution times using the *fetch-constrained dataflow-height (FCDH)*, a composite metric that captures the effects of both data-dependences and limited fetch bandwidth. For conventional dataflow height calculations, the input height (DH_{in}) of an instruction represents the time at which the instruction becomes data-ready and is computed as the maximum of the output heights (DH_{out}) of those instructions on which it is data dependent. DH_{out} — the time at which the instruction is complete — is computed by adding the instruction's execution latency to its DH_{in}. FCDH takes fetch into account by including a fetch constraint (FC) in the input height calculation. $FCDH_{in}$ is the maximum of the output heights ($FCDH_{out}$) of an instruction's dataflow predecessors *and* this fetch constraint — it represents the earliest time at which the instruction is both data-ready *and* fetched. Since instruction distances from the trigger are always lower in a data-driven context, it is this fetch term which promises that a given computation will execute at least as fast as a DDT as it would in a control-driven thread. The fetch constraint (FC) can be calculated in several ways; the simple one we use divides the instruction's dynamic distance from the trigger by the available fetch bandwidth. $FCDH_{out}$ is computed in the usual way — by adding the instruction's latency to its $FCDH_{in}$. The FCDH of a DDT is the maximum $FCDH_{out}$ of its instructions.

In Figure 3, we compute the FCDH for the DDT from our running example. For simplicity, we assume that all operations have unit latencies. The figure shows two separate FCDH calculations: one for the control-driven context and one for the data-driven context. Each computation is represented by three columns — I# is the instruction's dynamic distance from the trigger, FC is computed by dividing I# by the available fetch width, FCDH is computed using dataflow-rules and FC. Our calculation is for a 4-wide machine so, for the control-driven calculation, we use a fetch width of 4. However, to simulate the fact that a DDT *shares* fetch bandwidth with the main thread, the data-driven calculation uses a fetch width of 2.

The FCDH computation shows where a DDT gets its performance advantage. In the control-driven context, instructions are further away from the trigger. While the DDT fetches the two instances of *I10* in the cycle immediately following triggering and executes them in cycles 2 and 3, the main thread must wait until cycle 3 to fetch the first instance of *I10* and until cycle 6 to fetch the second one. By that time, the DDT has fetched and executed the entire computation. On a 4-wide machine, this DDT accelerates the critical results by 4 cycles. Repeating these calculations for an 8-wide machine (not shown), we find that doubling available fetch bandwidth does not help the DDT but greatly helps the control-driven context for which critical computation execution time drops from 9 to 6 cycles. DDTs attack the control-driven thread's sequential fetch constraints — they themselves are not fetch bound. Wider fetch is a brute force attack on the same problem.

The FCDH metric is intuitive. However, it oversimplifies the control-driven model by ignoring sources of control-driven fetch under-utilization and the nature of the other work in the main thread. It oversimplifies the data-driven model, ignoring contentious parallel execution. Unfortunately, more accurate estimates require detail equivalent to full simulation.

		Control-Driven			Data-Driven		
		I#	FC	FCDH	I#	FC	FCDH
I10	ldq r1, 0(r1)	0	0	1	0	0	1
I10	ldq r1, 0(r1)	11	3	4	1	1	2
I10	ldq r1, 0(r1)	22	6	7	2	1	3
I2	ldq r2, 8(r1)	25	7	8	3	2	4
I3	beq r2, I10	26	7	9	3	2	5
I5	ldt f1, 16(r2)	28	7	9	5	3	5

FIGURE 2. Calculating FCDH for a DDT.

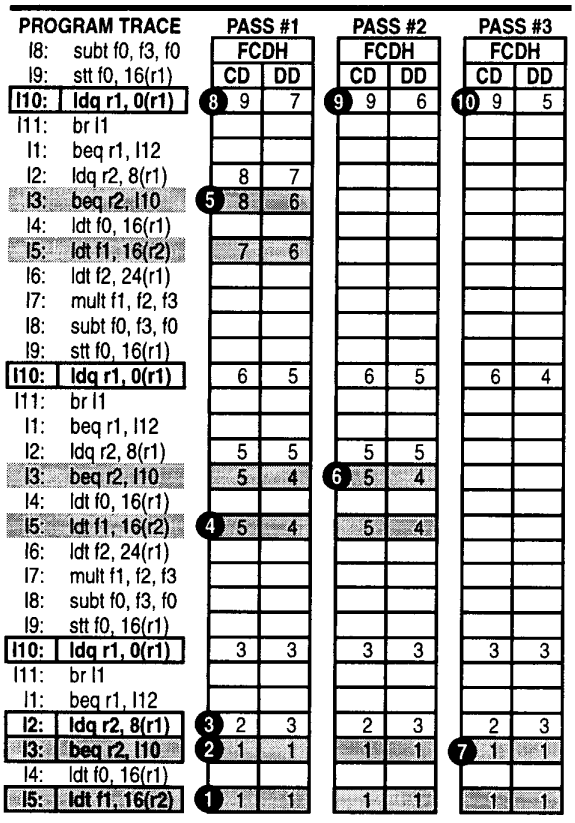


FIGURE 3. DDT Selection Algorithm. The algorithm makes three bottom-to-top passes, each time including fewer critical instruction instances, until it finds a DDT that provides a sufficient execution (FCDH) advantage.

3.2 Extracting Threads from a Program Trace

We now present an algorithm for extracting static DDT annotations from program traces. The algorithm is illustrated in Figure 3. Input to the algorithm is a program trace on which all critical load instances that miss in the cache and all mispredicted critical branch instances are marked. The trace is also marked with all load and store address which are needed to establish memory dependences.

The algorithm constructs a DDT candidate for every misbehaving instance of a critical instruction (marker 1). A DDT candidate is built by walking backwards through the trace and adding any instruction that (a) is itself a misbehaving instance of a critical instruction (marker 2) or (b) satisfies an active register or memory dependence for an included instruction, i.e. writes to a register or a memory location that is read by an instruction in the DDT candidate (marker 3). When an instruction is added, the dependence it satisfies is deactivated and its own input dependences are added.

Adding instructions to DDTs is simple. The key to the algorithm is figuring out when to stop. Essentially, we want to find the DDT for which a data-driven engine has the largest execution time advantage over a control driven engine. The FCDH was designed to answer this exact question. Whenever we add an instruction to our DDT candidate, we compute the FCDH of the newly created DDT (i.e., the DDT that assumes the new instruction is the trigger) in both the control-driven and data-driven contexts. If the difference between the control-driven and data-driven FCDH is greater than any other such previously computed difference, the new instruction becomes the trigger (marker 4) and the search continues for what might be an evem better DDT (marker 5).

The search for ever better DDTs must itself stop at some point. We use two criteria to determine when to stop the search. First, the algorithm considers a *trace window* of only the 1024 most recent instructions. The reason for this restriction is that a DDT instructions that are moved too far ahead of their architectural location in the program consume storage for too long before they are eventually integrated. As it turns out, the bulk of a computation for a given result happens close to that result [30] so this particular constraint is rarely activated. The second criterion, a hard limit on the total number of instructions in the DDT, is activated much more frequently. The purpose of this restriction is to limit DDT overhead.

To give thread selection the best chance to succeed, the thread constructor repeats the selection process for each dynamic instance multiple times, each time allowing fewer critical instructions to be added along the way. In the figure, the algorithm makes three passes over the trace, adding critical instructions from three iterations (marker 5), then from two (marker 6), and finally from one (marker 7), building a new DDT each time (markers 8, 9 and 10, respectively). This enhancement leads to the discovery that including the critical branch and load from a single iteration gives a better DDT (marker 10) than including computations from multiple iterations (markers 8 and 9).

When the program trace is completely processed, the DDT candidates are pruned based on relative frequencies. Thread selection can be tuned using maximum DDT size, trace window size, control- and data- driven width parameters and FCDH difference threshold.

This algorithm finds fairly good DDTs. *Optimal* DDT selection is a combinatorial problem where the inclusion of a computation in a DDT must consider not only the resource impact on the main thread but also the fetch delay induced in other computations in the DDT. DDT overlap, overhead, and likelihood of eventual integration are additional factors that our algorithm accounts for only implicitly.

4 Hardware Implementation

In this paper, we propose and present an implementation of DDMT that builds on a simultaneous multithreading (SMT) [27, 29] processor like Compaq's announced Alpha 21464 [7]. In addition to multiple sequencers, an SMT processor has a centralized implementation that allows resources to be flexibly shared among several threads. Flexible resource allocation lowers the run-time cost of DDTs by letting them "steal" whatever bandwidth the main thread is unable to exploit. It also allows us to conduct a fair evaluation of DDMT by constructing DDMT and non-DDMT systems with identical resource and bandwidth budgets. The shared physical register file and register renaming also support integration. We do not rule out other implementations. Other multithreaded microarchitectures, including decentralized ones like chip multiprocessing (CMP) [13], can support DDMT, albeit with a different implementation of integration or perhaps no integration at all.

4.1 Life Cycle of a Data Driven Thread

Figure 4 shows an SMT pipeline enhanced to support DDMT. The shaded structures and bold paths are DDMT-specific additions and modifications. Shaded slots in conventional structures indicate that DDT instructions may occupy these structures. The important events in the life of a DDT and its instructions are marked and numbered.

A DDT is dynamically "born" when an instance of its trigger instruction is renamed in a control-driven thread (marker 1). Implicit in this statement is the restriction that DDTs cannot launch other DDTs. This measure is also taken in an attempt to contain overhead since the alternative could potentially create a DDT explosion. Trigger instructions can be recognized using either pre-decode bits or a small lookup table. The decision regarding which pipeline stage should fork DDTs is actually fairly important. Forking a DDT early maximizes its fetch advantage over the control-driven main thread and the latency it can "absorb." However, early forking also increases the probability that the trigger instruction is itself mis-speculated and that the DDT is falsely forked. Empirically, forking DDTs at fetch produces many false DDTs that, although they can be detected and aborted, consume a significant amount of fetch bandwidth. In our implementation, DDTs are forked at the rename stage, reducing false squashes by roughly 50% while delaying DDTs by a relatively small fixed amount. Reducing this overhead much further requires waiting for the trigger instruction to retire to fork DDTs and potentially incurring much longer delays.

A forked DDT must be initialized with an execution context. This context is a copy of the control-driven register map as it appears immediately after the renaming of the trigger instruction (marker 2). As mentioned in Section 2, this copy allows the DDT to pick up external values using register renaming and synchronizes the DDT and main thread with a common mapping that will later be used to start the integration process.

In parallel with the map copy operation, the thread controller begins scheduling the new DDT for fetch (marker 3). In our implementation, instructions from only one thread are fetched in any cycle. The thread control uses a modified ICOUNT [26] policy to decide which thread has control of fetch in a given cycle. The chosen thread is the one that has the fewest total entries in the instruction fetch queue (IFQ) and reservation stations (RS). Reorder buffer (ROB) occupancy is not used because DDT instructions are not allocated ROB entries.

DDT instructions are fetched out of the data-driven thread cache (DDTC) (marker 4) and placed into the IFQ. Upon exit from the IFQ, DDT instructions are renamed and entries for them are created in the integration table (IT) (marker 5). These actions are, in fact, not particular to DDT instructions. All instructions are renamed, and all are created new entries in the IT — unless an entry for them is already found in the IT which is the subject of the Section 4.2.

After renaming, DDT instructions are sent to the out-of-order execution engine (marker 6) where they are allocated reservation-station slots (RS). However, unlike control-driven instructions, DDT instructions are not allocated reorder buffer (ROB) slots, nor are they entered into the memory ordering buffers (LDQ, STQ). DDT instructions do not affect architectural state until they are integrated. In-order retirement does not need to be enforced for them because they do not really retire. For a unified physical register file organization, like that of our SMT, the ROB also controls the freeing of physical registers — an instruction's retirement frees the physical register previously mapped to its output. DDT instructions — as long as they have not been integrated — do not share these semantics. When DDT instructions are allocated physical registers, the physical registers previously mapped to these locations either belong to control-driven instructions, in which case the control-driven ROB is responsible for freeing them or by previous instructions in the DDT, in which case we don't want to free them — we want to integrate them! In lieu of the ROB, the integration table (IT) controls the freeing of registers allocated by DDT instructions.

DDT instructions issue from the reservation stations like conventional instructions, reading from the same pool of physical registers and executing on the same functional units (marker 7). The only difference is in the handling of DDT loads and stores. DDT loads and stores cannot really be ordered or disambiguated and hence they have no need to occupy slots in the memory ordering queues. However, there is a need for a mechanism to provide store-to-load forwarding for DDT instructions, without sending DDT stores to the cache. We use a variation of the speculative memory cloaking mechanism [12] for this purpose. DDT stores are entered into a buffer where their address/value pairs are marked with a DDT identifier. DDT loads probe this structure in parallel with cache access, potentially picking up a value from a store with a matching address/identifier pair (marker 8).

Completed DDT instructions write their results into the physical register file and forward them to other waiting DDT instructions. When a completed DDT instruction is removed from the execution engine, the only remaining record of its existence is its IT entry which marks the physical register holding the result it computed. The second phase in the life of a DDT begins when the main thread that spawned it begins renaming the instructions corresponding to the DDT.

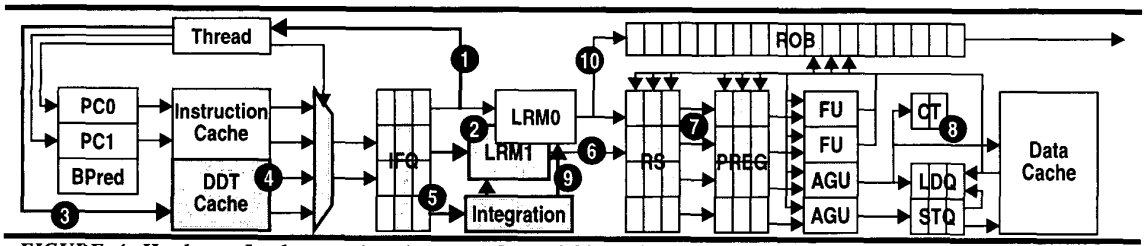


FIGURE 4. Hardware Implementation Aspects. Logical block diagram of a data-driven multithreading enabled SMT processor. In an actual implementation, the rename tables, instruction buffers and re-order buffers would be shared. They are distributed for illustration purposes. DDTC is the data-driven thread cache. CT is the cloaking table.

The working example in Figure 1 showed the integration process in detail. As the control-driven main thread renames instructions, it matches mappings found in its own map table with ones in the IT to locate the physical registers holding the results previously computed by corresponding DDT instructions. If a physical register is found, the main thread remaps it as the output of the current control-driven instruction (marker 9). The post-rename handling of an integrated instruction is exactly the opposite of that of a DDT instruction. An integrated instruction *is* entered into the ROB (marker 10) and memory ordering queues (LDQ,STQ) so that it can be ordered with respect to other control-driven instructions, but it is *not* allocated a new reservation station because it is either complete or already sitting in a reservation station allocated to it in its DDT execution. When a completed branch that was initially mis-predicted is integrated, recovery starts immediately, expediting the fetch of correct-path instructions and relieving some of the demand on the fetch engine.

4.2 Integration

The integration mechanism incorporates results from DDT instructions directly into the main thread, sparing the main thread the task of re-executing the work. With integration, a DDT can perform entire computations on behalf of the main thread. Without it, a DDT can impact performance only indirectly, for instance by prefetching. However, integration is more general than that — it is a mechanism for sharing and reusing results among the different execution contexts of a single program. Previous work has described integration as a mechanism for implementing squash reuse, salvaging results that were unnecessarily lost due to a sequential mis-speculation recovery [18]. The enabling principle for that application is the same one we exploit here. An instruction instance and its result are unambiguously identified by the identity of the creating instruction and the physical registers used as inputs to the operation. Any subsequent matching instruction that has the same input physical registers must, by definition, be a re-executed instance of the original instruction and can, by the same token, claim the result as its own. Integrating a result into a new execution context involves only updates to the appropriate context mappings. Integration does not require reading from or writing to the physical registers themselves.

Our previous work on integration-based squash reuse [18] describes the mechanics of integration in detail — the requirements of the base microarchitectures, the integration algorithm and circuit, the structure of the integration table, and a snooping based solution for guaranteeing the safe integration of loads. We will not go into these details here.

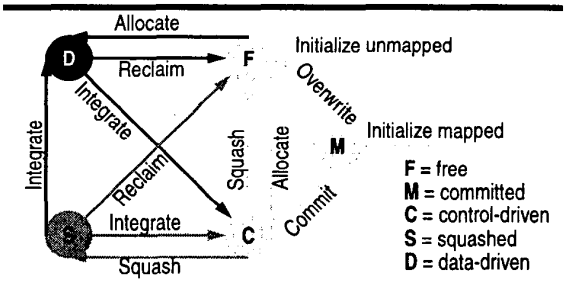
Now we describe a new underlying framework for integration that combines squash reuse with the DDT reuse that is a central feature of DDMT. Aspects of this framework are shown in Figure 5. One feature of this framework is that it allows integration to proceed in several directions. Obviously, control-driven threads may integrate both squashed results and DDT results. A natural extension of this is that DDT instructions can be integrated, then subsequently squashed and re-integrated! However, our framework also allows DDT instructions to integrate main thread results as well as results from other DDTs. These two capabilities are useful because they ensure that DDT results can still be integrated in situations where two DDTs partially overlap and in the event that the main thread *temporarily* runs ahead of a DDT.

In our framework, the integration table (IT) assumes the duties of the free list manager and contains entries for *all* physical registers, not only the ones that are integration candidates. Each physical register is in one of five states: *Free*, *coMmitted*, *Control-driven*, *Data-driven*, or *Squashed*. The states and the processor actions that effect the transitions between them are shown at the top of the figure. The *F*, *M*, and *C* states, shown in light gray, are the basic physical register states of a processor that does not implement integration. Registers in the *F* state are unmapped, those in the *M* state are mapped to the last committed instances of the architectural registers, and those in the *C* state are mapped to the outputs of the active instructions.

An implementation of integration-based squash reuse requires the addition of the *S* state and the corresponding dark gray transitions. Rather than transition into the *F* state on mis-speculation recovery, registers allocated to squashed instructions remain mapped and transition into the *S* state where they can subsequently be integrated back into the *C* state. Since the only pointer to a register in the *S* state may be in the IT, if it is not eventually integrated no execution event can trigger its freeing. To prevent leakage, a register can be spontaneously reclaimed from the *S* to the *F* state.

DDT integration requires the addition of the *D* state, which marks all registers that were allocated by DDTs and have yet to be integrated into the main thread, and the black transitions. Registers in the *D* state can also be spontaneously reclaimed to the *F* state, for reasons similar to those stated above.

The bottom of Figure 5 expands on the register transitions having to do with integration. Rows correspond to the type of instruction attempting to perform the integration, columns to pre-integration register state, and table entries to the post-integration register state. Registers in the *F* or *M* state cannot be integrated for obvious reasons. A register in the *C* state cannot be integrated by a control-driven instruction since that would create a situation with a single physical register mapped to two main thread instructions simultaneously! A *C* state register *can* be integrated by a DDT instruction, but it must remain in the *C* state to avoid the aforementioned case. Instructions in the *S* and *D* states can always be integrated and these assume the state of the integrating instruction.



	Physical Register State				
	F	M	C	S	D
Control-Driven Instruction	-	-	-	C	C
Data-Driven Instruction	-	-	C	D	D

FIGURE 5. Integration Framework. *(Top) Physical register state transition diagram. States and transitions belonging to conventional processors are in light gray. Integration based squash reuse integration adds those in darker gray. DDT integration adds those in black. (Bottom) Integration specific transition rules.*

5 Performance Evaluation

In this section, we evaluate the performance potential of DDMT in two roles — reducing the observed latency of first-level data cache misses and reducing the resolution latency of mispredicted branches. Our baseline system is an 8-wide SMT similar in spirit, but perhaps not in internal organization, to the Alpha 21464 [7]. The DDMT system also includes a DDTC and cloaking table. We do not model the effects of DDMT on processor cycle time or pipeline depth.

It is difficult, and perhaps improper, to simulate DDT integration — the integration of DDT instructions by the main thread — without simulating squash reuse integration as well. In order to correctly attribute performance to DDMT and DDT integration, our base processor configuration simulates integration-based squash reuse. For performance reporting purposes, the number of integrated DDT instructions is equal to the number of committed instructions whose physical register outputs were allocated by DDTs. DDT instructions that are integrated and squashed are not counted, while DDT instructions that are integrated, squashed, and integrated again (via squash reuse) are counted only once.

5.1 Methodology

The benchmarks we use are a collection of programs selected from the SPEC2000 and Olden pointer-intensive benchmark-suites. The Olden programs, *em3d, bh* and *mst*, are in essence microbenchmarks, executing a single algorithm on a synthetic input. Their relative simplicity allows correlations to be clearly drawn between DDT metrics and performance and trade-offs to be clearly explored. At the same time, their complexity is high enough to fully exercise the thread selection algorithm. Programs are selected for their (relatively) high branch misprediction or data-cache miss rates. 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 DDT selection algorithm permits a maximum DDT length of 32 instructions and considers only misbehaving instances as potential DDT candidate seeds. FCDH metrics are computed using control- and data- driven fetch widths of 8 and 4, respectively, and assuming a 10 cycle latency for critical loads, a 3 cycle latency for non-critical loads and a 1 cycle latency for all other instructions. The DDT selection phase uses a smaller and, where possible, different input data set than the DDMT performance measurement phase.

Our cycle-level simulator is built using the SimpleScalar 3.0 [3] Alpha toolkit. We model an SMT processor with full out-of-order speculative execution, register renaming, non-blocking caches, finite miss resources and cycle accurate bus utilization. We simulate 4 hardware contexts that share all bandwidths and resources. Our DDMT configuration executes a single control-driven thread and up to three concurrent DDTs. Table 1 shows the simulation parameters in detail.

Our base microarchitecture makes two concessions to DDMT. First, it has a large number of physical registers, 512, where a machine with 64 architectural registers and a 128-entry ROB would need only 196. Second, it supports a large number of outstanding cache misses, 64, where a machine with only 64 loads simultaneously in-flight typically provides for fewer. The extra registers hold DDT results as they wait to be integrated. The extra miss resources reflect the fact that DDMT can overlap many cache misses that cannot be overlapped in a purely sequential machine.

5.2 Targeting 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 can further increase memory-level parallelism (MLP) by overcoming sequencing constraints. Results for six benchmarks are summarized in Table 2. All the instruction quantities are dynamic and counted in millions of events. Execution time savings range from 4.1% for *parser* to 24.8% for *mst*.

The other metrics support these numbers. Load latency is the average difference between the issue and completion times of

Front End	16K entry combined 10-bit history gshare and 2-bit predictors. 2K entry, 4-way associative BTB. 3-stage fetch. 32-entry instruction buffer. Up to 8 instructions from two cache blocks fetched per cycle with a maximum of one taken branch per cycle. Up to 4 hardware contexts share the fetch engine on a cycle basis. Each cycle, instructions are fetched from the active thread with the fewest instructions in the fetch queue and reservation stations.
Execution Engine	8-way, out-of-order, speculative issue with a maximum of 128 instructions or 64 loads or 32 stores in-flight. 80 reservation-station slots. 512 physical registers. 2 cycle decode/register-rename. 2 cycle register read. Loads speculatively issue in the presence of earlier stores with unknown addresses. The load and subsequent instructions are squashed and refetched on a memory ordering violation. A 64-entry collision history table (CHT) synchronizes loads that mis-speculated in the past. 1 cycle address generation. 2 cycle store-to-load forwarding. Loads and branches have the highest scheduling priority. Scheduling 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, 2 cycle access, first level data cache. A maximum of 64 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 associative, 12 cycle access second level cache. 70-cycle memory latency. 16B per cycle bandwidth to the L2 cache and 8B per cycle bandwidth to main memory. Cycle level bus utilization modeled.
Execution Units	8 int ALU (latency = 1), 3 int mult (3), 3 int div (20), 4 FP add (2), 3 FP mult (4), 3 FP div (24), 4 load/store (3) . The FP adders and all multipliers are fully pipelined.
DDMT Support	16-entry, 1 cycle access data-driven thread cache (DDTC) with a maximum of 32 instructions per DDT. Up to 8 data-driven instructions from a single DDT fetched per cycle. 16-entry cloaking table.

TABLE 1. Simulated machine configuration.

every *committed* load. The baseline load latency for our processor is 3 cycles — 1 cycle to compute an address and 2 to either hit in the first level cache or the store-queue. However, with integration, average load latency can actually drop below this number. DDT loads that are integrated after completion and subsequently committed contribute an observed latency of zero. MSHR occupancy, a measure of MLP, is the per-cycle average number of in-flight cache misses. In all cases, DDMT reduces load latency while increasing miss overlapping. Ideally, DDMT would not generate any new and unnecessary cache misses, and the increase in MSHR occupancy simply reflects the same number of misses divided by a shorter execution time. This indeed appears to be the case in most of the benchmarks. One exception is *mcf* for which some unnecessary misses are generated.

The reduction in sequentially observed load latency does not always translate directly to performance. One factor is the natural latency tolerance that exists in programs. The latency of some loads may be naturally hidden by branch mispredictions, other long latency loads, or simply other parallel work. Attacking the, albeit long, latency of these loads will not improve performance. An interesting and important extension to this work would be to narrow its focus to only loads whose latency actually determines performance [24].

A more significant factor that limits performance is the overhead associated with DDTs. We approximate the full effect of DDT resource contention by comparing the number of instructions fetched by each system. Overall, the use of DDTs tends to increase the total number of instructions fetched by the system, albeit not significantly as the main thread typically fetches fewer wrong path instructions. There are two effects here. First, reducing load latency reduces the resolution time of branches that depend on these loads. The dominant effect, however, is simple fetch contention — the main thread fetches fewer instructions while waiting for branches to resolve.

One troubling statistic is the relatively small number of fetched DDT instructions that are eventually integrated. Table 2 lists integration counts broken down by the status of the DDT instruction at the time it was integrated. DDT instructions integrated after completing have done useful work on behalf of the main thread. Instructions integrated after having issued but before completing contribute some work, but empirically very few instructions are integrated in this state. Integrated instructions that have not issued are essentially useless, because they did not save the main thread any work. The integration of these instructions indicates that the DDT was *trying* to do the right thing, but didn't have sufficient time in which to do it. The integration rates we observe can sometimes be very high (*mst*), but are often lower than 30%. Integration rates for completed instructions are lower still, although most DDT instructions do complete by the time they are integrated. Two factors contribute to this inefficiency. First, our choice to fork DDTs at the rename stage unnecessarily forks a fair number of DDTs using trigger instructions that turn out to lie along mis-speculated paths. The more signifi-

			parser	mcf	gzip	vpr	em3d	mst
Instructions committed (M)			4203.56	259.62	3367.27	692.50	67.75	230.77
Base	Instructions fetched (M)		8358.33	529.39	5883.09	1304.95	124.64	232.46
	Load latency (cycles)		5.48	12.43	3.41	3.86	11.39	20.22
	MSHR occupancy (/cycle)		1.49	3.23	0.83	1.70	9.42	2.44
Base + DDMT	DDTs forked (M)		15.85	4.06	26.28	10.19	0.72	0.53
	Instructions fetched (M)	Ctrl-driven	8351.17	466.88	5203.62	1249.14	110.77	232.35
		Data-driven	380.18	108.08	671.02	151.80	21.17	15.97
	Instructions integrated (M)	Total	109.68	34.44	269.20	41.27	7.15	14.45
		Completed	105.73	32.01	248.52	40.81	4.99	9.70
	Critical loads integrated (M)	Total	15.60	6.18	22.43	5.36	3.87	4.09
		Completed	11.67	4.99	13.80	5.09	2.34	1.66
	Load latency (cycles)		4.20	7.45	3.11	3.36	8.30	14.80
	MSHR occupancy (/cycle)		1.49	3.81	1.05	1.87	10.80	3.14
	Execution time saved (%)		**4.1**	**9.2**	**15.4**	**12.0**	**15.9**	**24.8**

TABLE 2. Using DDMT to pre-execute loads that miss in the L1 cache.

			em3d-1	em3d-2	em3d-3	mst-1	mst-2	mst-3
Base + DDMT	DDTs forked (M)		0.72	0.68	0.69	0.53	0.53	0.53
	Instructions fetched (M)	Ctrl-driven	110.77	109.68	109.24	232.35	232.29	232.26
		Data-driven	21.17	20.49	21.32	15.97	17.56	19.15
	Instructions integrated (M)	Total	7.15	8.93	9.59	14.45	14.41	14.39
		Completed	4.99	7.00	8.73	9.70	11.60	12.35
	Critical loads integrated (M)	Total	3.87	4.92	5.31	4.09	4.08	4.07
		Completed	2.34	3.50	4.62	1.66	2.40	2.79
	Load latency (cycles)		8.30	6.05	3.90	14.80	11.71	9.76
	MSHR occupancy (/cycle)		10.80	11.36	11.88	3.14	3.62	4.02
	Execution time saved (%)		**15.9**	**21.1**	**23.1**	**24.8**	**37.5**	**44.9**

TABLE 3. Effect of induction variable unrolling on DDT latency tolerance and performance impact.

			eon	crafty	gzip	vpr	em3d	bh
Instructions committed (M)			458.29	4264.77	3367.27	692.50	67.75	977.44
Base	Instructions fetched (M)		710.58	7082.44	5883.09	1304.95	124.64	2336.08
	Branch mispredictions (M)		3.98	31.78	16.02	4.78	0.73	10.90
	Resolution latency (cycles)		15.35	16.21	29.94	47.48	28.90	39.28
Base + DDMT	Threads forked (M)		3.06	21.76	29.27	10.41	0.69	39.42
	Instructions fetched (M)	Ctrl-driven	687.23	6949.68	5228.01	1215.61	108.51	1943.91
		Data-driven	26.08	183.66	565.41	133.10	18.52	294.24
	Instructions integrated (M)	Total	6.11	28.38	312.58	41.19	3.00	29.69
		Completed	4.69	24.96	273.06	32.23	2.91	16.60
	Critical instr's integrated (M)	Total	7.12	3.60	18.11	3.49	0.31	5.52
		Completed	3.66	3.13	14.63	3.07	0.31	2.43
	Mispredictions integrated (M)	Total	0.29	0.74	4.79	0.69	0.12	2.37
		Completed	0.17	0.68	2.60	0.58	0.12	1.10
	Resolution latency (cycles)		14.81	16.02	21.39	44.51	18.94	32.98
	Execution time saved (%)		**1.2**	**0.4**	**12.9**	**5.3**	**11.5**	**7.1**

TABLE 4. Using DDMT to pre-compute outcomes of branches that are likely to be mispredicted.

cant factor, however, is inherent in the implicit control structure of DDTs themselves. A DDMT processor implicitly predicts upcoming instances of a critical instruction by the appearance of its DDT's trigger. Such speculation is not always correct — a dynamic instance of a trigger instruction does *not necessarily* imply that a dynamic instance of the corresponding critical instruction is forthcoming. However, absent control flow, a DDT cannot deduce this fact and is forced to execute all computations within it, even ones from which the main thread has diverged. Note that this is *not* the same as saying that DDT efficiency is tied to the branch prediction accuracy in the main thread — in fact, efficiency may increase with decreased prediction accuracy. However, while our DDTs and forking procedure are fundamentally speculative, no alternatives are obvious at this time.

Our example from Figure 1 showed a DDT that uses two instances of a loop's induction step (node = node->next) to overlap the control-driven execution of a computation with a DDT execution of one from two iterations ahead. This idiom, which we call *induction unrolling*, is quite powerful. By adding a single loop induction to the head of a DDT, we increase its latency-tolerance capability by one loop iteration. When the cost of an induction is low in terms of instructions and execution time, we have the flexibility to create DDTs that can tolerate nearly arbitrary latencies!

Table 3 shows the effectiveness of induction unrolling using the programs *em3d* and *mst*. Both programs have inexpensive induction operations — pointer-chases that hit in the cache. The table shows the performance impact of three DDTs for each program — using one, two, and three unrolled inductions, respectively. *Mst* exhibits the classic behavior. The number of instructions fetched by DDTs is progressively higher, mirroring the larger size of each DDT, while the number of integrated DDT instructions is constant. However, the number of *completed* instructions integrated grows with each additional induction, while average load latency decreases and total execution time decreases. *Em3d*'s behavior is similar. However, its variable loop iteration sizes mean that moving DDTs further ahead of their architectural iterations increases the likelihood that the control-driven thread will integrate the DDT and not vice versa.

5.3 Targeting Branch Mispredictions

Our second experiment uses DDTs to pre-compute the outcomes of frequently mispredicted branches. This application is particularly attractive because, in addition to expediting the fetch of correct-path instructions, it has the potential to directly reduce the number of instructions fetched along mispredicted paths and alleviate some of the fetch pressure created by DDMT itself.

Accelerated branch resolution results for six benchmarks are shown in Table 4. The branch misprediction resolution latency is calculated as the average number of cycles between the misprediction of a branch and its completion. For the DDMT system we also measure the number of mispredicted branches resolved by integrated DDT instructions. A mispredicted branch that integrates a completed DDT instruction is resolved instantly, for these DDMT accomplished its mission. Integration of non-completed branches may still improve performance if most of the *computation* of the branch has been completed by the DDT.

For the branch pre-computation application, we obtain execution time savings in the range of 0.4% for *crafty* to 12.9% for *gzip*. These savings are in proportion with the reductions in misprediction resolution latency. As projected, this application of DDMT sometimes reduces the total number of instructions fetched by the system. Not surprisingly, programs for which this is the case — *gzip*, *em3d* and *bh* — observe the greatest benefit.

Our work investigates the performance impact of integration-based branch resolution. With integration matching pre-executed DDT instruction instances with their corresponding dynamic main thread instances, the early resolution of integrated branches is an application that comes essentially for free. However, integration-based branch resolution is not perfect — it cannot resolve mispredictions earlier than the integration stage itself. At its best, integration-based resolution can lower the misprediction penalty to a constant fetch-resteering penalty plus the pipeline distance between prediction and renaming/integration. Technically, since branch outcomes are pre-computed in advance, they *could* be sent to their dynamic

		Cache Misses			Branch Mispredictions		
		mcf	**vpr**	**mst**	**eon**	**gzip**	**em3d**
Base	Instructions fetched(M)	529.39	1304.95	232.61	710.58	5883.09	124.65
	Load latency (cycle)	12.43	3.86	19.85	2.88	3.41	11.38
	Resolution latency (cycle)	24.57	47.48	279.54	15.35	29.94	28.90
Base + DDMT	Instructions fetched (M)	574.96	1400.95	248.32	713.32	5793.42	121.66
	Load latency (cycle)	7.45	3.36	14.80	2.88	3.24	10.34
	Resolution latency (cycle)	19.44	39.02	176.89	14.81	21.39	18.94
	Execution time saved (%)	**9.2**	**12.0**	**24.8**	**1.2**	**12.9**	**11.5**
Base + critical scheduling	Instructions fetched (M)	529.42	1305.15	232.63	710.11	5880.50	124.44
	Load latency (cycle)	12.41	3.80	19.85	2.88	3.41	11.37
	Resolution latency (cycle)	24.64	39.16	291.70	15.07	29.22	26.99
	Execution time saved (%)	**0.0**	**-0.1**	**-0.1**	**0.2**	**0.4**	**0.3**
Base + DDMT - Integration	Instructions fetched (M)	605.93	1406.29	248.33	721.35	6456.13	129.67
	Load latency (cycle)	9.45	3.65	14.78	2.91	3.34	10.75
	Resolution latency (cycle)	22.56	41.29	176.57	15.20	26.12	20.66
	Execution time saved (%)	**1.7**	**8.5**	**25.0**	**-0.3**	**1.4**	**8.8**

TABLE 5. Performance contributions of data-driven sequencing and integration.

instances earlier in the pipeline, perhaps as early as the branch predictor itself. In fact, several mechanisms for doing just that have already been proposed [4, 8, 17]. However, without the benefit of relying on dynamic data-dependences which are not available that early in the pipeline, these techniques must match pre-computations to dynamic branches in ad hoc ways that may actually introduce mispredictions! Integration-based resolution may not be able to lower the misprediction penalty as much as these mechanisms, but the nature of integration guarantees that it will not introduce any mispredictions.

5.4 Sequencing and Integration Contributions

In the introduction we claimed that data-driven sequencing, the ability to *sequence* through critical computations faster than a control-driven thread, is an important performance enabling aspect of DDMT — simply prioritizing critical computations while using conventional sequencing is insufficient. We also maintained that integration is important, especially for order-sensitive uses of DDMT like early branch resolution. We now quantitatively support these claims by attributing the performance of DDMT to each factor. The results are reported in Table 5. For space reasons, we include results for a *total* of six benchmarks, three from each initial experiment.

To measure the importance of data-driven sequencing, we simulate the program using the DDT annotations as *prioritization hints* to the scheduler rather than as templates for actual DDTs. As we suspected, critical-computation priority-scheduling in hardware is largely ineffective and sometimes harmful. Prioritizing parts of critical computations that are already in the instruction window does not increase parallelism and may delay execution of the oldest instructions in the machine, preventing them from retiring and freeing up slots for future potentially critical computations. DDMT's power arises from its ability to expose parallelism by using data-driven sequencing to increase the effective scheduling window.

To quantify the contribution of integration, we repeat our experiments but do not integrate the work performed in the DDTs. Squash reuse integration is still performed. 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 prefetching effect and integration provides only modest additional gains. Integration is more important for fast branch resolution, however, because every cycle added to the misprediction penalty directly delays the processing of future correct path instructions. Accelerated branch resolution does not provide much (*gzip*) if any (*eon*) benefit in the absence of integration. One Table 5 data point, *em3d*, seems to contradict this assertion. The effect we observe here stems from the fact that *em3d*'s branch computations contain, via data-dependences, loads that miss in the cache. The branch computations prefetch the loads which in turn expedite dependent branch execution in the main thread.

The synergy of DDMT and integration extends beyond branch resolution. In fact, it is the presence of integration that oftentimes makes certain kinds of DDTs profitable. In particular, induction unrolling relies heavily on integration, specifically DDT-to-DDT and main thread-to-DDT integration. Consider a DDT that unrolls 5 induction copies. If a copy of this DDT is forked at every iteration, then each dynamic main thread induction actually resides in 5 DDTs. Without integration, each DDT would have to compute its 5 induction steps before reaching the actual computation of interest — a process that would take at least 5 cycles or more if the induction is of the pointer-chasing variety or consists of multiple instructions. Repeating induction steps that have been computed by previous DDTs not only wastes bandwidth, but it also robs the DDT of much of its execution advantage. If the amount of work in each main thread iteration is small, induction unrolling may not be profitable. With integration, each DDT can leverage the induction steps already computed by previous DDTs, so that instead of executing 5 induction steps, a DDT *integrates* 4 and executes 1. In fact, integration guarantees that the induction chain and, as a result, the actual computations of interest are being computed as fast as possible. *Mcf* illustrates this phenomenon well. *Mcf*'s main DDT uses induction unrolling to expedite the execution of loads in a tight loop. Although it only performs prefetching, this DDT requires integration to achieve most of its effect. Without integration, overlapping DDTs are unable to leverage each other's induction computation, reducing each DDT's ability to tolerate latency.

6 Related Work

Long latency loads and mispredicted 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. This same approach to parallelism formed the basis for the explicit dataflow architectures [2, 5, 10, 11, 14] of the 1970s. Dataflow architectures expose computation structure at the architectural interface to enable theoretical levels of parallelism and latency tolerance. Speculative data-driven multithreading (DDMT) 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.

Access/execute decoupling [21] is another technique for disentangling computations from one another and allowing load computations, which as a group are considered more critical, to proceed unimpeded by unrelated processing. DDMT can be thought of as a microarchitectural decoupling of computations that result in cache misses.

Other architectures and microarchitectures use *control-driven speculative threads* to accelerate sequential programs. *Speculative control-driven multithreading (CDMT)* — commonly known as speculative multithreading — systems include the Multiscalar architecture [9, 22], single-program speculative multithreading (SPSM) [6], thread-level data-speculation (TLDS) [25], and dynamic multithreading (DMT) [1]. In CDMT, 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. CDMT and DDMT are diametrically opposed in their philosophies. CDMT exploits primarily *thread-level parallelism (TLP)*. For CDMT techniques to succeed, the program must be *partitioned* into *sequentially contiguous, mutually disjoint* threads that are parallel or at least nearly so. 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 (ILP)*. Data-driven threads perform implicit control speculation, follow data-dependences closely, and are automatically load balanced by an SMT processor. DDMT does not require the presence of TLP to be effective. DDTs have the flexibility to attack individual latencies and points of performance degradation. The price for flexibility is that instructions in data-driven threads must be refetched and re-renamed by a control-driven thread.

Assisted execution [23] and SSMT [4] are techniques for accelerating a sequential program by using idle thread contexts to execute auxiliary code that prefetches data, presets the branch predictor or performs some other performance enhancing task. In contrast with DDMT, the auxiliary code is *not* an annotated part of the original program that can later be integrated. 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 the true sense, able to impact performance only indirectly but unable to contribute values directly to the main program thread.

Dependence based prefetching and branch pre-execution [16, 17] and the branch-flow microarchitecture [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 reuse the results to one degree or another. Dependence-based prefetching uses post-retirement 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 adds an ad-hoc correspondence mechanism to allow pre-computed branch results to be matched with dynamic branch predictions. The branch-flow microarchitecture is a different implementation of the same idea.

7 Summary and Future Work

Data-driven multithreading (DDMT) is a new speculation model. Speculation is performed at the granularity of a *data-driven thread (DDT)*, 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 DDT become available, the DDT is microarchitecturally forked and executed in parallel with the sequential program. While the main thread sequentially fetches the entire program, the DDT deals with only those instructions that constitute the critical computation of interest. Consequently, it typically fetches and executes these computations much faster than would be possible in 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 mispredicted are chosen as DDTs. By pre-executing these performance critical instructions, DDTs "absorb" latencies that would otherwise impact sequential performance.

An interesting component of DDMT is integration, a facility for incorporating results produced in DDTs into the main thread, avoiding the need for re-execution. Integration works by using dataflow relationships to prove that a DDT instruction and a main thread instruction actually correspond to the same dynamic instruction instance and allow the main thread to take possession of the physical register holding the DDT instruction's result. Integration enables DDTs to perform actual work on behalf of the main 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 includes register renaming and the flexible resource allocation policies needed to run DDTs efficiently. The additional hardware required to support DDMT includes a data-driven thread cache (DDTC) to hold the DDTs, a cloaking table to implement DDT store-to-load forwarding, and integration logic.

Our experiments show that DDMT can be used to reduce performance degradations due to cache misses and branch mispredictions. 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 microarchitectural gadgets.

There are several avenues for future work. Our implicit control DDT model is simple and disallows runaway threads. One of its drawbacks is its inability to abort a DDT computation that diverges from the main thread. This capability appears important to limiting DDT overhead, and models that incorporate explicit control should be investigated.

In this paper, we present a DDMT implementation based on a simultaneous multithreading (SMT) microarchitecture — a choice made for reasons of both implementation simplicity and pending availability. However, other architectures and microarchitectures that support multiple threads and even speculative control-driven threads are on the horizon as well. Mapping DDMT or pre-execution onto these 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. Better data-driven threads will enhance the performance impact of DDMT. Certainly, an implementation of our or any other algorithm in hardware would ease the acceptance of DDMT by moving the technique into the purely microarchitectural realm.

Acknowledgements

This work was supported in part by National Science Foundation grants MIP-9505853 and CCR-9900584, donations from Intel Corp. and Sun Microsystems, the University of Wisconsin Graduate School and an Intel Ph.D Fellowship. The authors thank the anonymous referees for their reviews and Adam Butts, Milo Martin, Dan Sorin, and Craig Zilles for their comments on various incarnations of this manuscript.

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] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin-Madison, Madison, WI 53706, Nov. 1993.

[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] 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.

[13] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-Chip Multiprocessor. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, Oct. 1996.

[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] E. Rotenberg, S. Bennett, and J. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *Proc. 29th International Symposium on Microarchitecture*, pages 24–35, Dec. 1996.

[16] 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.

[17] 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.

[18] A. Roth and G. Sohi. Register Integration: A Simple and Efficent Implementation of Squash Re-Use. In *Proc. 33rd Annual International Symposium on Microarchitecture*, Dec. 2000.

[19] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. Technical Report CS-TR-00-1414, University of Wisconsin, Madison, Mar. 2000.

[20] A. Roth and G. Sohi. Speculative Data Driven Sequencing for Imperative Programs. Technical Report CS-TR-00-1411, University of Wisconsin, Madison, Feb. 2000.

[21] J. Smith. Decoupled Access/Execute Computer Architecture. In *Proc. 9th International Symposium on Computer Architecture*, Jul. 1982.

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

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

[24] S. Srinivasan and A. Lebeck. Load Latency Tolerance in Dynamically Scheduled Processors. In *Proc. 31st International Symposium on Microarchitecture*, pages 148–159, Nov. 1998.

[25] 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*, Feb. 1998.

[26] 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.

[27] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proc. 22nd International Symposium on Computer Architecture*, pages 392–403, Jun. 1995.

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

[29] W. Yamamoto and M. Nemirovsky. Increasing Superscalar Performance Through Multistreaming. In *Proc. 1995 Conference on Parallel Architectures and Compilation Techniques*, Jun. 1995.

[30] C. Zilles and G. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proc. 27th International Symposium on Computer Architecture*, pages 172–181, Jun. 2000.