University of Rhode Island
Dept. of Electrical and Computer Engineering
Kelley Hall
4 East Alumni Ave.
Kingston, RI 02881-0805, USA

Technical Report No. 032002-0101

Realizing High IPC Through a Scalable Memory-Latency Tolerant Multipath Microarchitecture

A. Khalafi, D.A Morano, D.R. Kaeli
{akhalafi, dmorano, kaeli}@ece.neu.edu
Department of Electrical and Computer Engineering
Northeastern University
and
A.K. Uht
uht@ele.uri.edu
Department of Electrical and Computer Engineering
University of Rhode Island

2nd April 2002

This work has been submitted for publication.

Abstract

This paper explores a microarchitecture that achieves high execution performance on conventional single-threaded program codes without compiler assistance. Microarchitectures that can have several hundreds of instructions simultaneously in execution can provide a means to extract larger amounts of instruction level parallelism, even from programs that are very sequential in nature. However, several problems are associated with such microarchitectures, including scalability issued related to control flow and memory latency.

We present a basic overview of our microarchitecture and discuss how it addresses scalability as we attempt to execute many instructions in parallel. We also show how we use multipath execution to limit the impact of conditional branch mispredictions. We provide simulation results for several geometries of our microarchitecture that illustrate how high IPC can be realized from integer programs. We also explore algorithms that dynamically reassign speculative paths, reallocating hardware resources to higher priority paths. Finally, we present data that shows the tolerance of the microarchitecture to high memory latency.

1 Introduction

A number of studies into the limits of instruction level parallelism (ILP) have been promising in that they have shown that there is a significant amount of parallelism within typical sequentially oriented single-threaded programs (e.g., SpecInt-2000). The work of researchers like Lam and Wilson [13], Uht and Sindagi [20], Gonzalez and Gonzalez [7] have shown that there exists a great amount of instruction level parallelism

This work was partially supported by the National Science Foundation through grants MIP-9708183 and EIA-9729839; by the URI Office of the Provost; and by the Spanish Ministry of Education. Patents applied for. A version of this work has been submitted for publication. Copyright 2001, the authors.

(ILP) that is not being exploited by any existing computer designs. Unfortunately, most of the fine-grained instruction level parallelism inherent in integer sequential programs spans several basic blocks. Data and control independent instructions, that may exist far ahead in the program instruction stream, need to be speculatively executed to exploit all possible inherent ILP. A large number of instructions need to be fetched each cycle and executed concurrently in order to achieve this. We need to find the available program ILP at runtime; we need to provide sufficient hardware to find, schedule, and otherwise manage the out-of-order speculative execution of control and data independent instructions.

The relatively small instruction fetch windows present on existing processor designs cannot span the program instruction space necessary to effectively exploit the available instruction-level parallelism. A microarchitecture with a large instruction fetch window and execution window offers the possibility to both expose (fetch) and exploit (execute) the available ILP. For those computer applications that demand the highest IPC possible on integer codes, a microarchitecture that is able to execute possibly several hundreds of instructions speculatively is needed in order to maximize execution performance.

A fundamental challenge is how to find program parallelism and then allow execution to occur speculatively and out of order over a very large number of instructions. Of course, the microarchitecture has to also provide a means to maintain the architectural program order that is required for proper program execution. It is also usually very desirable to support legacy instruction set architectures (ISAs) when pursuing high IPC. For this reason, we want to explore a microarchitecture that does not impact the ISA.

We present a novel microarchitecture in this paper that can be applied to any existing ISA. Our microarchitecture is targeted at obtaining substantial program speedups on integer codes. The microarchitecture can speculatively execute hundreds of instructions ahead in the program instruction stream and thus expose large amounts of inherent ILP. We use multipath execution to cover latencies associated with branch mispredictions. We also take advantage of control and data independent instructions through our use of execution-time predication. Finally, the microarchitecture is also substantially insensitive to memory component latencies even though it requires higher memory bandwidth than more conventional machines.

The rest of this paper is organized as follows. Section 2 presents related work on high-IPC machines and on multipath execution. Section 3 presents our proposed microarchitecture. Section 4 presents simulation results for a range of machine configurations, and shows the potential impact of multipath execution when applied. We also discuss how our microarchitecture reduces our dependence on the memory system by providing a large amount of local caching on our datapath. Finally, we summarize and conclude in section 5.

2 Background

There have been several attempts at substantially increasing program IPC through the exploitation of ILP. The Multiscalar processor architecture [18] is another attempt at realizing substantial IPC speedups over convention superscalar processors. However, our approach is quite different than theirs and their approach relies on compiler participation where we do not. A notable attempt at realizing high IPC was done by Lipasti and Shen on their Superspeculative architecture [14]. They achieved an IPC of about 7 with realistic hardware assumptions. The Ultrascalar machine [10] achieves *asymptotic* scalability, but only realizes a small amount of IPC due to its conservative execution model. Nagarajan et al proposed a *Grid Architecture* of ALUs connected by an operand network [1]. This has some similarities to our work. However, unlike our work, their microarchitecture relies on the coordinated use of the compiler along with a new ISA to obtain higher IPCs.

Early work on multipath execution was dominated by IBM in the late 1970s and 1980s [5]. The earliest attempts at multipath execution started with the ability to prefetch down both outcomes of a conditional branch. This became more aggressive to the point of actually executing down both outcomes of a conditional branch. This has been explored in work such as that by Wang [23]. More aggressive research by Uht and Sindagi [20] explored the intersection of both multipath execution and future large-scale microarchitectures capable of possibly hundreds of instructions being executed simultaneously. They also addressed the general question of speculatively executing more than two paths simultaneously. Work on dual path execution (only two speculative paths) has been done by Heil and Smith [9]. Klauser et al explored multipath execution (including more than two speculative paths) on the PolyPath microarchitecture [12].

Exploring multipath execution in the context of simultaneous multithreading (SMT) has been done by Wallace et al [22]. Ahuja et al [2] explore some limits for speedups from multipath execution but their work

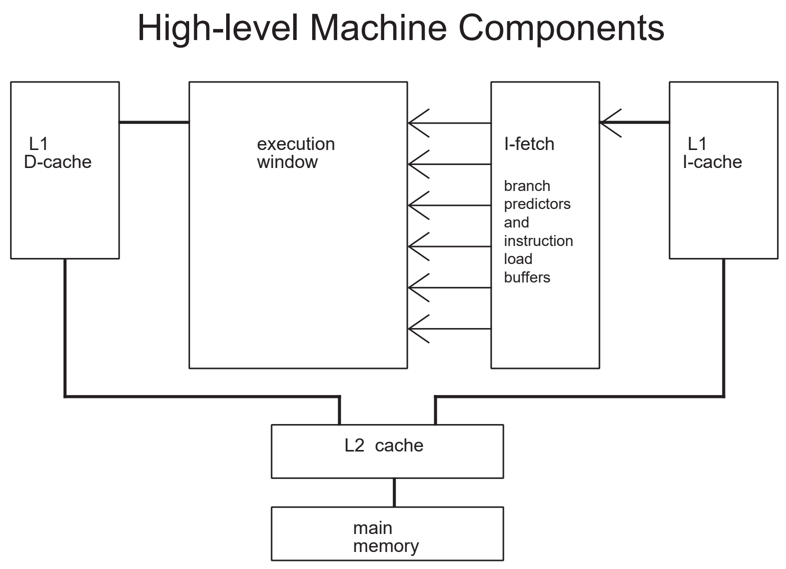
High-level Machine Components
L1 D-cache
execution window
I-fetch
branch predictors and instruction load buffers
L1 I-cache
L2 cache
main memory

Figure 1: *High-level View of the Distributed Microarchitecture.* Shown are the major hardware components of the microarchitecture. With the exception of the execution window block, this is similar to most conventional microarchitectures

.

is still largely restricted to more conventional (modest sized) microarchitectures. Our present work explores the use of multipath execution on a significantly larger scale than that of Ahuja or most previous work.

3 Microarchitecture Description

The microarchitecture is very aggressive in terms of the amount of speculative execution it performs. This is realized through a large amount of scalable execution resources. Resource scalability of the microarchitecture is achieved through its distributed nature along with repeater-like components that limit the maximum bus spans. Contention for major centralized structures is avoided. Conventional centralized resources like a register file, reorder buffer, and centralized execution units, are eliminated.

The microarchitecture also addresses several issues associated with conditional branches. Spawning alternative speculative paths when encountering conditional branches is done to avoid branch misprediction penalties. Exploitation of control and data independent instructions beyond the join of a hammock branch [6] is also capitalized upon where possible. Choosing which paths in multipath execution should be given priority for machine resources is also addressed by the machine. As shown by Uht and Sindagi [20], equal priority to all simultaneous paths of a program is not the most efficient use of hardware resources. The predicted program path is referred to as the *mainline* path. We give execution resource priority to this mainline path with respect to any possible alternative speculative paths. Since additional speculative paths have lower priority with respect to the mainline path, they are referred to as *disjoint* paths. The term *disjoint* refers to that fact that the assignment of execution resources for that path is likely (and should likely) be deferred in time as compared with when execution resources are assigned to the mainline path. This sort of strategy for the spawning of alternative speculative paths results in what is termed *disjoint eager execution* (DEE). This is in contrast to *singlepath* speculative execution (widely used at the present) or *eager execution.* We therefore refer to disjoint paths as simply *DEE paths.* These terms are taken from Uht's 1995 work [20]. More detailed information about this microarchitecture can be found in a technical report by Uht et al [21].

3.1 High-Level Microarchitecture Components

Figure 1 provides a high-level view of our microarchitecture. Our microarchitecture shares many basic similarities to most conventional machines. The main memory block, the L2 cache (unified in the present case),

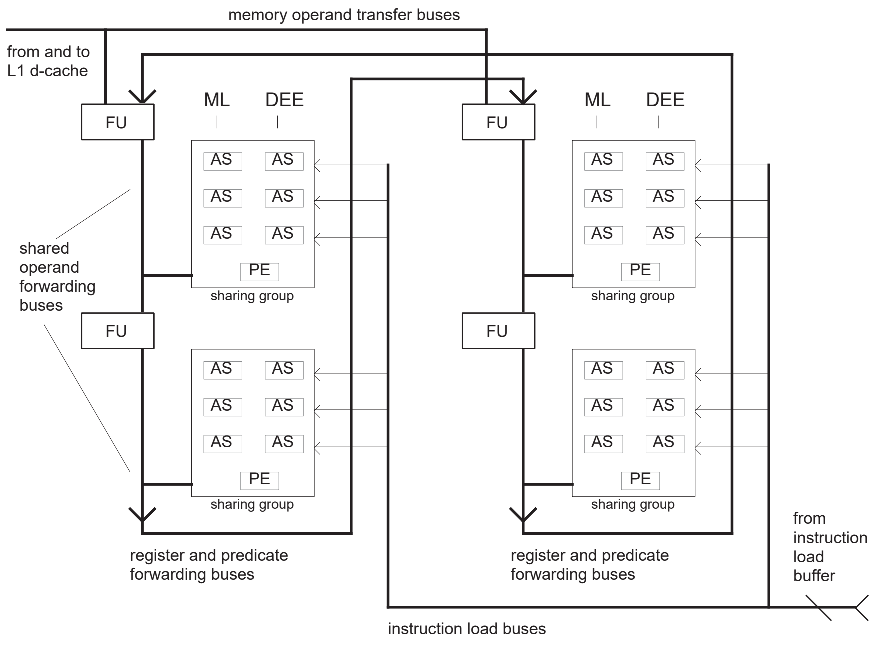
memory operand transfer buses
from and to L1 d-cache
FU
ML
DEE
AS
PE
sharing group
shared operand forwarding buses
register and predicate forwarding buses
from instruction load buffer
instruction load buses

Figure 2: *High-level View of the Distributed Microarchitecture.* Shown is a layout of the Active Stations (AS) and Processing Elements (PE) along with some bus interconnections to implement a large, distributed microarchitecture.

and the L1 instruction cache are all rather similar to those in common use. Except for the fact that the main memory, L2 cache, and L1 data cache are all address-interleaved, there is nothing further unique about these components. Our L1 data cache is similar to most conventional data caches except that it also has the ability to track speculative memory writes. Our L1 d-cache shares a similar goal with the Speculative Versioning Cache [8] but is simpler is some respects. Since we allow speculative memory writes to propagate out to the L1 data cache, multiple copies of a speculative write may be present within the L1 data cache at any time. They are differentiated from each other through the use of time-tags. Time-tags are the basic mechanism used in the microarchitecture to order all operands, including memory operands, while they are being used by instructions currently being executed. Time-tags are small values that are associated with operands that serve as both an identifying tag and as a means to order them with respect to each other. The Warp Engine [4] also used time-tags to manage large amounts of speculative execution, but our use of them is much simpler than theirs.

The i-fetch unit first fetches instructions from i-cache along one or more predicted program paths. Due to our relatively large instruction fetch bandwidth requirement, we allow for the fetching of multiple i-cache lines in a single clock. Instructions are immediately decoded after being fetched. All further handling of the instructions is done in their decoded form. Decoded instructions are then staged into an *instruction load buffer* so that they are available to be loaded into the *execution window* when needed. The execution window is where our microarchitecture differs substantially from existing machines. The transfer of decoded instructions from the instruction load buffer to our adaptation of reservation stations is termed *instruction load.* This instruction load buffer is organized so that a large number of instructions can be broadside loaded into the execution window in a single clock. The multiple buses going from the i-fetch unit to the execution window in Figure 1 is meant to reflect this operation. This maximum number of instructions loaded into the execution window at a time is termed the *column height* of the machine.

3.2 The Execution Window

Figure 2 shows a more detailed view of the execution window with its subcomponents. We have extended the idea of Tomasulo's reservation station [19] to provide the basic building block for a distributed microarchitecture. Tomasulo's reservation station provided for the simultaneous execution of different instructions over several functional units. Register results from the functional units were placed on a common data bus and looped back to provide source register operands for instructions waiting in the reservations stations as well as for updating of the register file. In our microarchitecture, an output result is not looped back to the input of the same reservation station that provided the result but rather is forwarded to different stations that are spatially separated, in silicon or circuit board space, from the first. This operation is termed *operand forwarding*. We call our adaptation of the reservation station an *active station* (*AS*). Like a reservation station, an active station can only hold a single instruction at a time. After an instruction is loaded into as AS, it remains there until it can be retired (either committed or squashed) from the execution window. However, instructions can be re-executed when events indicate that a re-execution is warranted or required for proper program order fulfillment.

Rather than lay the ASes out in silicon simply next to functional units that will execute the instructions loaded to them (like with the original reservation station idea), we lay them out in a two dimensional grid whereby sequentially loaded instructions will go to sequential ASes down a column of the two dimensional grid of ASes. The use of a two dimension grid simply provides a means to implement the necessary hardware either in a single silicon IC or through several suitable ICs laid out in a grid on a circuit board. The number of ASes in the height dimension of the grid is the same as the column height of the machine, introduced previously. The example machine of Figure 1 has a column height of six (six instruction load buses shown feeding the execution window). The column height can also be seen more clearly in Figure 2 when the total number of ASes in a single column of ASes is counted.

Dispersed among the active stations are associated execution units. An execution unit is represented in the figure as a *processing element* (*PE*). PEs may consist of an unified all-purpose execution unit capable of executing any of the possible machine instructions or, more likely, consist of several functionally partitioned units individually tailored for specific classes of instructions (integer ALU, FP, or other), as is typical of most current machines. Groups of active stations along with their associated processing element are termed a *sharing group* (SG). They are termed sharing groups because the execution resources within one of them can be shared among the enclosed ASes. The example machine of Figure 2 consists of two columns of SGs. Sharing groups somewhat resemble the relationship between the register file, reorder buffer, reservation stations, and function units of most conventional microarchitectures. They have a relatively high degree of bus interconnectivity between them, as conventional microarchitectures do. The ASes serve the role of both the reservation station and the reorder buffer of more conventional machines. The transfer of a decoded instruction, along with its associated operands, from an AS to its PE is isolated to within the given SG. The use of this execution resource sharing arrangement also allows for reduced interconnections between adjacent SGs. Basically, only operand results need to flow from one SG to subsequent ones.

In our present microarchitecture, we always have two columns of ASes within a SG. The first AS column is reserved for the main-line path of the program and is labeled *ML* in the figure. The second column of ASes is reserved for the possible execution of a DEE path and is labeled *DEE* in the figure. In this machine example, each SG contains three rows of ASes (for a total of six) and a single PE. Many machine sizes have been explored so far but only a subset of these sizes is further investigated in this paper. A particular machine is generally characterized using the tuple:

- sharing group rows
- active station rows per sharing group
- sharing group columns
- number of DEE paths allowed

These four characteristic parameters of a given machine are greatly influential to its performance, as expected, and is termed the *geometry* of the machine. These four numbers are usually concatenated so that the geometry of the machine in Figure 2 would be abbreviated `2-3-2-2`.

When an entire column of ASes is free to accept new instructions, generally an entire column worth of instructions are loaded to the free AS column from the instruction load buffer, in a single clock. Conditional branches are predicted just before they are entered into the instruction load buffer (for subsequent load to the ASes). The prediction of a branch accompanies the decoded instruction on an instruction load operation.

Also employed within the execution window is a scheme to dynamically predicate, at execution time, all instructions that have been loaded into active stations. This predication scheme essentially provides for each loaded instruction an *execution predicate*. These execution predicates are just a single bit (like with explicit architectural predication) but are entirely maintained and manipulated within the microarchitecture itself, not being visible at the ISA level of abstraction.

3.3 Operand Forwarding and Machine Scalability

An interconnect fabric is provided to forward result operands from earlier ASes to later ASes, in program order. Result operands are one of three possible types: register, memory, and instruction execution predicates. The interconnect allows for arbitrary numbers of sharing groups to be used in a machine while still keeping all bus spans to a fixed (constant) length. All of the buses in Figure 2, with the exception of the instruction load buses, form the interconnection fabric. Several bus arrangements are possible but we only further explore one such arrangement (that shown in the figure). In the general case, several buses are used in parallel to make up a single forwarding span. This is indicated by the use of the bold lines for buses in the figure. More than one bus in parallel for each bus span is generally required to meet the bandwidth needs of the machine.

Active bus repeater components are used (and required) to allow for constant length bus spans. A bus repeater component is generally termed a *forwarding unit* (FU) and is so labeled in the figure. These forwarding units do more than just repeat operand values from one span of a bus to the next. For registers and memory, operands are filtered so that redundant forwards of the same value (as compared with that last forwarded) are eliminated. These can also be termed *silent forwards*. This filtering provides a means to reduce the overall bandwidth requirements of the forwarding interconnection fabric. Each forwarding unit employed in the present work also has a small amount of storage for memory operands. This storage serves as a cache for memory operand values. We term this small cache storage a *L0 data cache*.

For register and predicate operands, values that are generated by ASes contend for one of the outbound buses (labeled *shared operand forwarding buses* in the figure) to forward the value. Requests for bus use will be satisfied with any bus clock-slot that may be available on any of the buses in parallel, belonging to a given span. All other ASes on the outbound bus span snoop operand values forwarded from previous (in program order) ASes. In addition, a forwarding unit (the bus repeater) also snoops the same operands and forwards the operand value to the next bus span if necessary (if the value was different than the previous value). For register and predicate operands, they are also looped around from the bottom of one column of SGs to the top of the next column of SGs. Operands from the bottom of the far right column of SGs gets looped around to the top of the far left column. This behavior forms the characteristic ring pattern of operand flow, inherent in many microarchitectures. Forming a closed loop with these buses, and essentially just renaming columns (identifying the one closest to retirement), is easier than physically transferring (shifting) the contents of one column to the next when a column of ASes retires.

For memory operands, we employ a second operand forwarding strategy. When memory operands are generated by ASes, again the AS contends for one of the outbound buses (labeled *shared operand forwarding buses* in the figure) in order to forward the operand value. However, unlike the register and predicate operand forwarding strategy, memory operands also travel backwards, in program ordered time, and get snooped by the forwarding units that are at the top of each SG column. This is done so that the operand can be transfered onto a *memory operand transfer bus*, shown at the top of Figure 2. These buses are address-interleaved and provide the connectivity to get memory operands (generally speculative) over to the L1 data cache. Values are tentatively stored in the L1 data cache along with their associated operand time-tags until a committed value is determined. Similarly, operands returning from the L1 data cache to service requests from ASes, are first put on one of the memory operand transfer buses (based on the interleave address of the operand). These operands then get snooped by all of the forwarding units at the top of each SG column, after which the operand is forwarded on a shared operand forwarding bus to reach the requesting ASes.

Persistent register, predicate state and some persistent memory state is stored in the forwarding units. Persistent state is not stored indefinitely in any single forwarding unit but is rather stored in different units as the machine executes column shift operations (columns of ASes get retired and committed). However, this is all quite invisible to the ISA. This microarchitecture also implements precise exceptions [17] similarly to how they are handled in most speculative machines. A speculative exception (whether on the main-line path or a DEE path) is held pending (not signaled in the ISA) in the AS that contains the generating instruction until

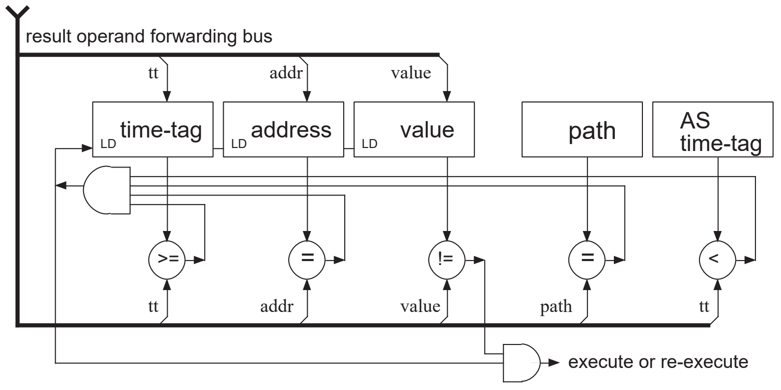

Figure 3: *Operand snoop logic within an AS.* The logic used for snooping of input operands for ASes is shown.

it would be committed. No action is needed for pending exceptions in ASes that eventually get squashed. When an AS with a pending exception does commit, the machine directs the architected control flow off to an exception handler through the defined exception behavior for the given ISA. This might include saving the precise instruction return address to either an ISA architected register or memory. Typically, the exception handler code will save the architected registers to memory using normal store instructions of the ISA. Interrupts can be handled in more flexible ways than exceptions. One way to handle interrupts is to allow all instructions currently being executed within the execution window to reach commitment, then architected program flow can vector off to a code handler, similarly as the case of instruction exceptions above.

3.4 Enforcing Program Order and Dependencies

Program dependencies (control, register, and memory) are maintained through the use of time-tags. Time-tags are associated with all transient operands within the machine. This has some resemblance to register tags used in more conventional microarchitectures but has been more generalized for use in this distributed microarchitecture. Since instructions remain in the ASes that they were loaded into until they retire, the whole set of ASes fulfill the role of the reorder buffer or register update unit of more conventional microarchitectures. As a column of ASes gets retired, that column becomes available for the loading of newly decoded instructions. In addition, a time-tag associated with each column get decremented. Time-tags associated with operands can be decomposed into row and column parts. The column part of the operand time-tag is identically the column time-tag, so when a column has its time-tag decremented, it effectively renames the operands within that column. The next column in the machine (with the next higher time-tag) becomes the next column that will get retired. The operation of decrementing column time-tags in the execution window is termed a *column shift*. The hardware used for the snooping of an input operand of an AS is shown in Figure 3. Basically, a new operand is snarfed when it has the same address and path identifier as the current AS as well as a time-tag value that is less than that of the current AS itself but greater or equal to that of the last snarfed operand. Simpler snooping hardware is used in forwarding units. A more detailed discussion of the mechanism used for enforcing program dependencies can be found in a report by Kaeli et al [11].

3.5 Conditional Branches and Multipath Execution

If a conditional backward branch is predicted taken, the i-fetch unit will speculatively follow it and continue loading instructions into the execution window for the mainline path from the target of the branch. This case allows for the capture of program loops within the execution window of the machine and can be thought of as hardware loop unrolling. For a backward branch that is predicted not-taken, we continue loading instructions following the not-taken output path. If a forward branch has a near target, such that it can be loaded within the execution window, then we load instructions following the not-taken output path of the branch, whether or not it is the predicted path. This represents the fetching of instruction in the memory or *static* order rather than the program dynamic order and is very common in the absence of a loop. The fetching and loading of instructions following the not-taken output path (static program order) of a conditional branch is very advantageous for capturing hammock styled branch constructs. Simple single-sided hammock branches generally have near targets, they are thus captured within the execution window.

Our mainline path continues along the predicted branch output path regardless of whether it was the taken or not-taken one. We spawn a DEE path for the opposite output of the branch from the mainline path case, whatever it is. For forward branches with a far target, if the branch is predicted taken, we load instructions following the target of the branch. If the branch is predicted not-taken, we continue loading instructions for the mainline path following the not-taken outcome of the branch. In both of these cases, we do not spawn a DEE path for this branch.

DEE paths are created by loading an available column of ASes, a free second column of ASes within a SG column, with the same decoded instructions from the AS column that contained the conditional branch instruction that gave rise to the DEE path. However, there are a limited number of AS columns available at any one time for DEE paths in the machine. Several strategies for the management of the DEE path AS columns are possible and we present two such strategies in the present work.

One such strategy is very simple and just spawns DEE paths for suitable conditional branches (as explained above) on a first come, first served basis. That is, as machine execution resources that are reserved for DEE paths become available, those resources are assigned to the next conditional branch for which a DEE path has not already been assigned. The DEE path is allowed to remain executing until its originating conditional branch gets resolved, at which point either it or the corresponding main-line path is squashed (the other of the two paths commits).

A second strategy that we explore is to monitor the amount of time that existing DEE paths have been resident in the execution window, and to squash those paths that have reached a residency threshold. Residency time in the execution window is approximated by the number of column shifts that have occurred since a DEE path was spawned. Again, a column shift corresponds with the retirement of the AS column with the lowest valued time-tag. The rationale is that the longer a DEE path is resident within the execution window, the likelihood of the conditional branch that gave rise to it being mispredicted becomes less. Thus, that DEE path is no longer using the underlying execution resources as well as maybe some new DEE path might (remember that the DEE path is used to follow the not predicted path). When a DEE path is squashed in order to make room for the spawning of a new DEE path, the associated executed resources are released and assigned to another conditional branch. This is termed the *release* strategy of managing DEE paths.

4 Simulation Results

We first describe our simulation process. Then results showing IPCs for each of three modes of execution of the machine is given. Finally, results showing the sensitivity of our machine to varying the latencies of several components in the memory hierarchy is presented.

4.1 Methodology

The simulator is a recently built tool that shares some similarity to SimpleScalar [3] but which was not based on it. We execute SpecInt-2000 and SpecInt-95 programs on a simulated machine that features a MIPS-1 ISA along with the addition of some MIPS-2 and MIPS-3 ISA instructions. We are using the standard SGI Irix system libraries so we needed to also support the execution of some MIPS-2 and MIPS-3 instructions (present in the libraries). All programs were compiled on an SGI machine under the Irix 6.4 OS and using the standard SGI compiler and linker. Programs were compiled with standard optimization (`-O`) for primarily the MIPS-1 ISA (`-o32`). No changes to the SGI compiler or linker were made to create the binary benchmark programs.

We chose five benchmark programs to work with, four from the SpecInt-2000 benchmark suite and one from the SpecInt-95 program suite. These programs were chosen to get a range of different memory and looping behavior, while also presenting challenging conditional control flow behavior. The particular programs used along with some statistics are given in Table 1. All programs were executed using the SpecInt reference inputs. All accumulated data was gathered over the simulated execution of 500 million instructions, after having skipped the first 100 million instructions. The first 100 million instructions, however, were used to warm up the various simulator memory caches. The dynamic conditional branches in Table 1 are a percent of total dynamic instructions. The numbers for the Simple Single-sided (S-S) Hammock branches [6, 16] are percentages of total dynamic conditional branches.

Table 1: Benchmarks Programs Simulated and Some Statistics.

benchmark	bzip2	parser	go	gzip	gap
br. prediction accuracy	90.5%	92.6%	72.1%	85.4%	94.5%
avg. L1-I hit rate	97.2%	96.6%	92.4%	94.7%	89.0%
avg. L1-D hit rate	98.8%	99.0%	98.8%	99.8%	99.3%
avg. L2 hit rate	90.1%	86.0%	96.8%	73.0%	88.5%
dynamic cond. brs.	12.0%	11.0%	12.1%	13.4%	6.5%
S-S hammock brs.	23.4%	42.1%	35.7%	45.2%	27.3%

Table 2: Machine geometries simulated for each of the benchmark programs.

SG rows	ASes per SG	SG columns	max D-paths
8	4	8	8
8	8	8	8
16	8	8	8
32	2	16	16
32	4	16	16

4.2 IPC Results for Singlepath and Multipath Execution

In this section, we present IPC data for three different modes of machine execution, each on five machine geometries. The execution modes are : no multipath execution (singlepath), simple multipath execution , and enhanced multipath execution. The numbers of each of the major machine components, for each of the five simulated geometries, are given in Table 2. Although we have explored a large number of various sized machines, these particular geometries were chosen in order to get a range of IPC performance across a number of very different machine sizes and shapes. The common machine characteristics used in this section for obtaining IPC results is given in Table 3. The L1, L2, and main memory access latencies do not include the forwarding unit and forwarding bus delays within the execution window. These machine characteristics are fairly representative of existing typical values for a 2 GHz processor. They are similar to, or more conservative than, a recent Pentium-4 (0.13 um) processor [15]. The IPC results for each of the three modes of machine execution are presented in Tables 4, 5, and 6. The geometry labels (4-tuples) at the tops of these tables consist of the concatenated numbers of machines components for: SG rows, AS rows per SG, SG columns, and the number of DEE paths allowed for that execution.

Table 4 gives the IPC results for each of the benchmark programs, when executing in singlepath mode. For singlepath execution, the allowed number of DEE paths is zero.

In addition to the individual benchmark IPC results, we also present the harmonic mean of the IPC across all benchmarks. Table 5 gives the results using the simple method (first-come-first-serve) for the spawning of DEE paths.

Examining the data, we can see that our simple (first-come-first-serve) multipath execution strategy significantly outperforms singlepath execution. This was expected as the microarchitecture is able to capture inside of the execution window many of the instructions following both outcomes of forward conditional branches, thereby largely hiding any misprediction penalties of those branches.

Next we want to get results for machine simulations using our more complicated *release* management strategy for DEE paths. In this enhanced strategy, DEE paths are allowed to remain within the execution window for a certain number of columns shifts (described previously). We first attempt to determine the best number of column shifts to use as a metric for path residency. For this we took one machine geometry and varied the number of column shifts the machine waits before releasing a DEE path. We use the machine

Table 3: *General machine characteristics.* These machine parameters are used for all simulations as the default except where one of these parameters may be varied.

L1 I/D cache access latency	1 clock
L1 I/D cache size	64 KBytes
L1 I/D block size	32 bytes
L1 I/D organization	2-way set associative
L2 cache access latency	10 clocks
L2 cache size	2 MBytes
L2 block size	32 bytes
L2 organization	direct mapped
main memory access latency	100 clocks
memory interleave factor	4
forwarding unit minimum latency (all)	1 clock
forwarding-bus latency (all)	1 clock
number of forwarding buses in parallel	4
branch predictor	PAg
	1024 PBHT entries
	4096 GPHT entries
	saturating 2-bit counter

Table 4: IPC results for singlepath execution.

geometry	8-4-8-0	8-8-8-0	16-8-8-0	32-2-16-0	32-4-16-0
bzip2	3.4	4.2	4.8	4.2	4.7
parser	2.8	3.3	3.8	3.5	3.9
go	2.6	3.2	3.6	3.4	3.6
gzip	3.4	4.4	5.2	4.8	5.3
gap	4.5	5.6	6.1	6.5	6.3
HAR-MEAN	3.2	3.9	4.6	4.2	4.6

Table 5: IPC results for multipath execution using the simple D-path strategy.

geometry	8-4-8-8	8-8-8-8	16-8-8-8	32-2-16-16	32-4-16-16
bzip2	3.9	4.8	5.5	4.9	5.3
parser	4.0	4.3	5.1	4.6	5.1
go	4.6	5.6	6.4	5.9	6.3
gzip	4.7	5.9	6.8	6.2	6.8
gap	5.6	6.9	7.1	8.3	7.2
HAR-MEAN	4.5	5.4	6.1	5.7	6.1

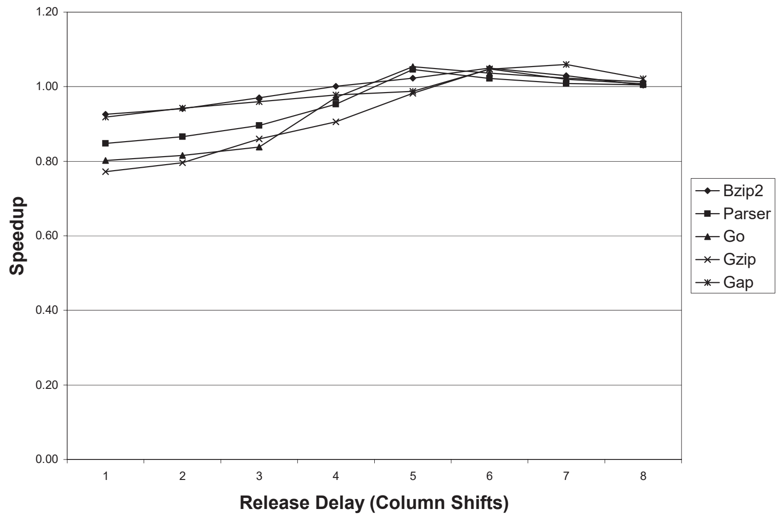

Figure 4: Machine IPC speedup results for varying column shifts.

Table 6: IPC results for multipath execution using the release D-path strategy.

geometry	8-4-8-8	8-8-8-8	16-8-8-8	32-2-16-16	32-4-16-16
bzip2	4.2	5.0	5.8	5.4	5.7
parser	4.3	4.6	5.3	5.0	5.4
go	5.1	5.9	6.7	6.5	6.8
gzip	5.0	6.3	7.0	6.7	7.2
gap	6.0	7.5	7.5	8.9	7.9
HAR-MEAN	4.8	5.7	6.4	6.3	6.5

geometry of 16-8-8-8 along with the parameters listed in Table 3 to investigate IPC speedups as the number of column shifts waited for is varied. Figure 4 shows the resulting IPC speedups, as compared with the simple disjoint path management strategy. For this particular machine geometry, this strategy performs worse than the simple strategy when the number of column shifts waited for is approximately fewer than four or five. However, when DEE paths are retained in the execution window for approximately six columns shifts, a performance speedup of about two to five percent is realized over the simple DEE path strategy. From these results, we will chose to allow DEE paths to remain in the execution window for six column shifts for all five machine geometries explored and all benchmarks.

Table 6 gives the results using the *release* method for the spawning of DEE paths. Again, these allow DEE paths to stay resident in the execution window through six column shifts (determined above) and the remaining parameters of the machine are those in table 3.

From these results, it is observed that our more complicated DEE *release* strategy performs better than our simple strategy for all machine geometries explored here. This is encouraging and suggests that attention to the management of the DEE paths is important to getting higher IPC speedups. Some other strategies for the management of how and when to spawn DEE paths have been proposed but they have not yet been explored. This is an area that we intend to explore more fully in the future.

Our lowest performing machine geometry (8-4-8-8) when executing in singlepath mode, yielded a harmonic mean IPC of 3.2. However, the same geometry machine, when executing using the enhanced DEE path strategy, yielded a harmonic mean IPC of 4.8 (substantially better). Finally, the largest sized machine geometry simulated using the enhanced DEE path strategy, yielded a harmonic mean IPC of 6.5.

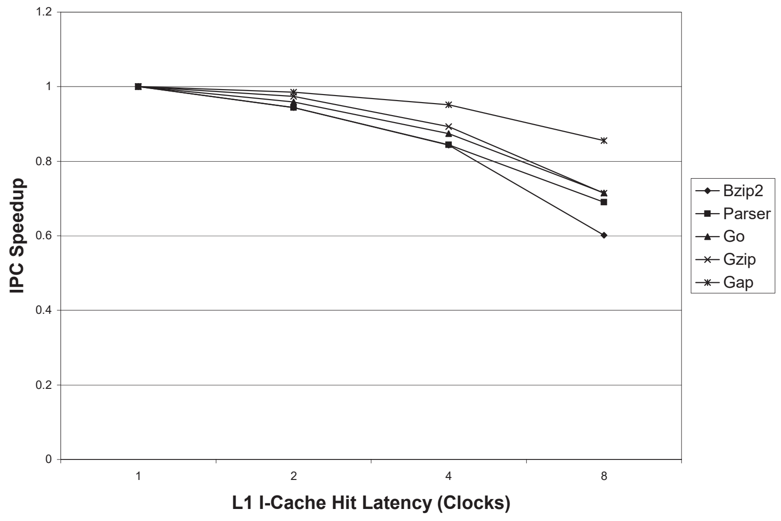

Figure 5: Machine IPC speedup results for varying L1 I-cache hit delay in clocks.

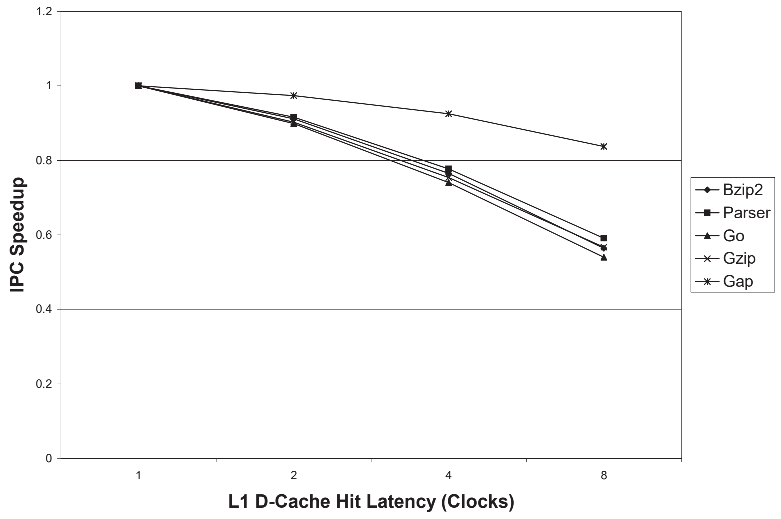

Figure 6: Machine IPC speedup results for varying L1 D-cache hit delay in clocks.

4.3 Memory Sensitivity Results

In this section we present IPC data corresponding to varying some parameters associated with the memory subsystem. We show IPC speedup results for varying the access latencies, in clocks, for: L1 I-cache, L1 D-cache, L2 cache, and main memory. All of this data was gathered on a machine geometry of 16-8-8-8 with the other parameters (the ones that are not varied) being those that are listed in Table 3. Figure 5 presents the IPC speedups for varying the L1 I-cache hit latency clocks from one up to eight. Figure 6 presents IPC speedup results as the L1 D-cache hit latency is also varied from one to eight clocks. For these two figures, the speedups (the ordinate) range from 0.0 through 1.2. Figure 7 presents the IPC speedup results as the L2 cache (unified I/D) hit latency is varied from one up to 16 clocks (our design choice was 10 clocks). Finally Figure 8 presents the IPC speedup results as the main memory access hit latency is varied from twenty clocks up to 800 clocks. For the L2 cache and main memory sensitivity graphs, the speedups (ordinate) range from 0.9 through 1.02. All IPC speedups in these figures are relative to the 1 clock access latency cases. As can be seen from Figures 5 and 6, the machine is slightly more sensitive to L1 D-cache latency than to L1 I-cache latency. This is expected due to the variability in speculative memory accesses that is present for data memory accesses, that is not as prevalent in instruction accesses. Fortunately, latencies of 1 or 2 clocks for

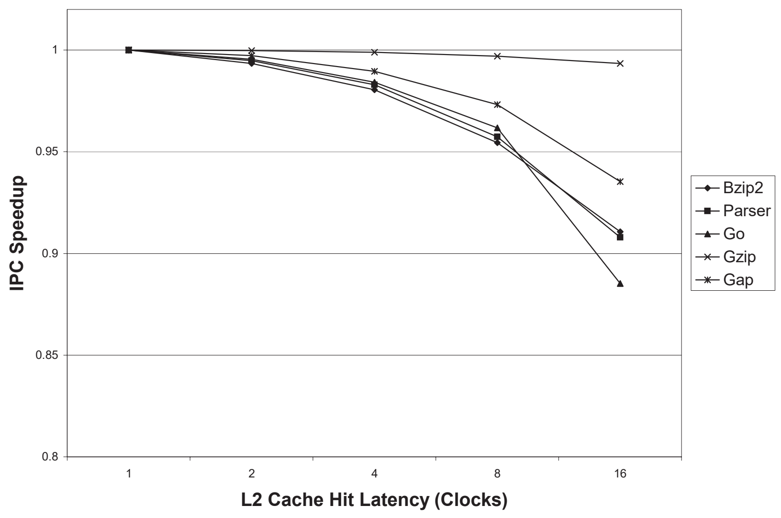

Figure 7: Machine IPC speedup results for varying L2 cache hit delay in clocks.

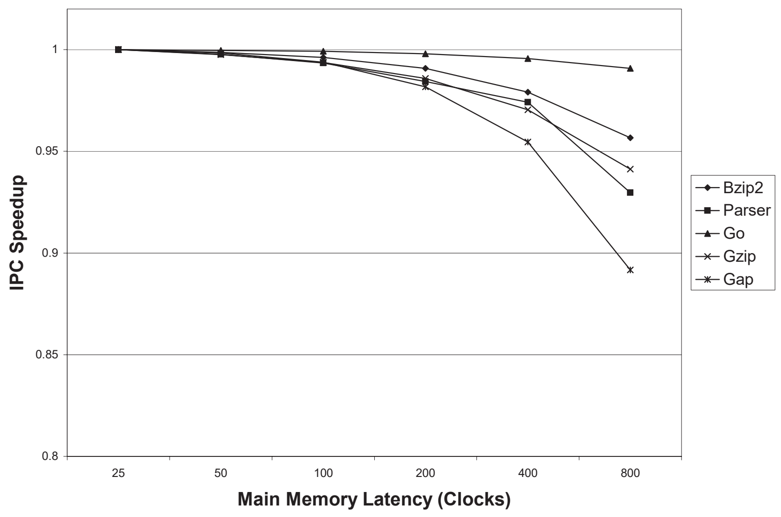

Figure 8: Machine IPC speedup results for varying main memory access latency in clocks.

L1 caches is more likely to scale better with increasing processor clock rate than memory components further from the processor.

For the L2 cache (ours is currently unified), speedups are possible when using short clock latencies than the 10 clocks that we used as our default. Although L2 latencies are likely to also scale somewhat with future increasing processor clock rates, they are not likely to scale as well as L1 is expected to do. Fortunately our machine already obtains good IPC numbers for an L2 latency of 10 clocks.

With respect to main memory, the microarchitecture is quite insensitive to latencies out to 100 clocks and only then starts to degrade slightly after that. Since 100 clocks (as we count it – after our repeater and bus delays) is probably typical at the present time (assuming a 2 GHz CPU clock rate and the latest DDR-SDRAMs), our memory system arrangement is properly hiding most of the long main memory latency as it should. Since our machine is still quite insensitive to main memory latency out to 800 clocks, we might expect to operate the current machine up to about 10 GHz with similar performance. Our insensitivity to main memory latency is due to both the conventional use of L1 and L2 caches but also to the width of our execution window. When memory load requests are generated from instructions soon after they were loaded into the execution window, the width of the machine (in SG columns) provides substantial time to allow for those memory load requests to be satisfied, even when they have to go back to L1, L2, and to main memory.

5 Conclusions

We have presented the overview of a large-scale distributed microarchitecture suitable for extracting high ILP from sequential programs. This microarchitecture is designed to also implement speculative multipath execution. We presented results for the machine executing in singlepath mode and two types of management for alternative speculative paths in multipath mode. It was shown that multipath execution provides additional IPC performance over singlepath execution using both of the heuristics that we explored for managing the alternative paths. Further, our enhanced DEE path *release* heuristic provided an additional average of about five percent better IPC over the simple DEE path management heuristic for a modest sized machine. We also showed that our microarchitecture exhibits significant insensitivity to a wide range of memory system component latencies.

References

[1] Nagarajan R., Sankaralingam K., Burger D., Keckler S.W. . A design space evaluation of grid processor architectures. In *Proceedings of the 34nd International Symposium on Microarchitecture*, New York, NY, Nov 2001. ACM Press.

[2] Ahuja P.S., Skadron K., Martonosi M., Clark D.W. Multipath execution: Opportunities and limits. In *Proceedings of the 12th International Conference on Supercomputing*, New York, NY, July 1998. ACM Press.

[3] Austin T.M., Burger D. SimpleScalar Tutorial . In *Proc. of MICRO-30*, Nov 1997.

[4] Cleary J.G, Pearson M.W and Kinawi H. The Architecture of an Optimistic CPU: The Warp Engine. In *Proceedings of the Hawaii Internationl Conference on System Science*, pages 163–172, January 1995.

[5] Conners W.D., Florkiowski J., Patton S.K. The ibm 3033: An inside look. *Datamation*, pages 198–218, 1979.

[6] Ferrante J., Ottenstein K., Warren J. The progream dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[7] Gonzalez J. and Gonzalez A. Limits on Instruction-Level Parallelism with Data Speculation. Technical Report UPC-DAC-1997-34, UPC, Barcelona Spain, 1997.

[8] Gopal S., Vijaykumar T.N., Smith J.E., Sohi G.S. Speculative versioning cache. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture*. IEEE, Feb 1998.

[9] Heil T.H., Smith J.E. Selective Dual Path Execution. Technical report, University of Wisconsin - Madison, Madison, WI, 1996.

[10] Henry D.S and Kuszmaul B.C. and Loh G.H. and Sami R. Circuits for Wide-Window Superscalar Processors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 236–247. ACM, June 2000.

[11] Kaeli D., Morano D.A., Uht A. Preserving Dependencies in a Large-Scale Distributed Microarchitecture. Technical Report 022002-001, Dept. of ECE, URI, Dec 2001.

[12] Klauser A.S., Pathankar A., Grunwald D. Selective eager execution on the polypath architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 250–259, New York, NY, June 1998. ACM Press.

[13] Lam M.S. and Wilson R.P. Limits of Control Flow on Parallelism. In *Proc. of ISCA-19*, pages 46–57. ACM, May 1992.

[14] Lipasti M.H and Shen J.P. Superarchitecture Microarchitecture for Beyond AD 200. *IEEE Computer Magazine*, 30(9), September 1997.

[15] Ludloff C. IA-32 Implementation, Intel P4. http://www.sandpile.org/impl/p4.htm, Jan 2002.

[16] Sankaranarayanan K., Skadron K. A scheme for selective squash and re-issue for single-sided branch hammocks. In *PACT: Work In Progress Session*, Oct 2001.

[17] Smith J.E., Pleszkun A.R. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, Sep 1988.

[18] Sohi G, Breach S., Vijaykumar T. Multiscalar processors. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pages 414–425. ACM, June 1995.

[19] Tomasulo R.M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, Jan 1967.

[20] Uht A. K. and Sindagi V. Disjoint Eager Execution: An Optimal Form of Speculative Execution. In *Proc. MICRO-28*, pages 313–325. ACM, Nov 1995.

[21] Uht A.K., Morano D.A., Khalafi A., de Alba M., Wenisch T., Ashouei M., and Kaeli D. IPC in the 10's via Resource Flow Computing with Levo. Technical Report 092001-001, Dept. of ECE, URI, Sept 2001.

[22] Wallace S., Calder B., Tullsen D.M. Threaded multiple path execution. In *Proceedings of the 25th International Symposium on Computer Architecture*, New York, NY, June 1998. ACM Press.

[23] Wang S.S.H, Uht A.K. Ideograph/ideogram: Framework/architecture for eager execution. In *Proceedings of the 23rd Symposium and Workshop on Microprogramming and Microarchitecture*, pages 125–134, New York, NY, Nov 1990. ACM Press.