Helper Threads via Virtual Multithreading On An Experimental Itanium® 2 Processor-based Platform

Perry H. Wang[1], Jamison D. Collins[1], Hong Wang[1], Dongkeun Kim[1†], Bill Greene[3]
Kai-Ming Chan[2], Aamir B. Yunus[2], Terry Sych[2], Stephen F. Moore[2], and John P. Shen[1]

Microarchitecture Research Lab, Microprocessor Technology Labs, Intel Corp.[1]
Enterprise Technology Enabling, Software Solutions Group, Intel Corp.[2]
PAL Technology, Enterprise Platforms Group, Intel Corp.[3]

ABSTRACT

Helper threading is a technology to accelerate a program by exploiting a processor's multithreading capability to run "assist" threads. Previous experiments on hyper-threaded processors have demonstrated significant speedups by using helper threads to prefetch hard-to-predict delinquent data accesses. In order to apply this technique to processors that do not have built-in hardware support for multithreading, we introduce virtual multithreading (VMT), a novel form of switch-on-event user-level multithreading, capable of fly-weight multiplexing of event-driven thread executions on a single processor without additional operating system support. The compiler plays a key role in minimizing synchronization cost by judiciously partitioning register usage among the user-level threads. The VMT approach makes it possible to launch dynamic helper thread instances in response to long-latency cache miss events, and to run helper threads in the shadow of cache misses when the main thread would be otherwise stalled.

The concept of VMT is prototyped on an Itanium® 2 processor using features provided by the Processor Abstraction Layer (PAL) firmware mechanism already present in currently shipping processors. On a 4-way MP physical system equipped with VMT-enabled Itanium 2 processors, helper threading via the VMT mechanism can achieve significant performance gains for a diverse set of real-world workloads, ranging from single-threaded workstation benchmarks to heavily multithreaded large scale decision support systems (DSS) using the IBM DB2 Universal Database. We measure a wall-clock speedup of 5.8% to 38.5% for the workstation benchmarks, and 5.0% to 12.7% on various queries in the DSS workload.

†Dongkeun Kim is currently a Ph.D. candidate in the Department of Electrical and Computer Engineering at the University of Maryland, College Park.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASPLOS'04, October 7–13, 2004, Boston, Massachusetts, USA.
Copyright 2004 ACM 1-58113-804-0/04/0010 ...$5.00.

Categories and Subject Descriptors

C.1.1 [**Processor Architectures**]: Single Data Stream Architectures—*Multiple-instruction-stream, single-data-stream processors (MISD)*
C.4 [**Performance of Systems**]: Modeling techniques

General Terms

Performance, Design, Experimentation

Keywords

Helper thread, cache miss prefetching, multithreading, switch-on-event, Itanium processor, PAL, DB2 database

1. INTRODUCTION

Helper threading is a performance optimization technique that works by launching subordinate threads [7, 36] to hoist the latency of certain critical computations on behalf of a main thread. Typically, this optimization has been exploited either to prefetch future data accesses [9, 10, 19, 20, 22, 23, 34, 41, 42, 43], or to precompute the outcome of future hard-to-predict branches [1, 8]. The majority of previous work has assumed a hardware multithreaded processor, and in particular, a simultaneous multithreaded (SMT) [38, 39] processor. Because of its ability to share hardware resources such as fetch bandwidth and execution functional units among threads, an SMT microarchitecture is well suited for overlapping execution of the main thread and its helper threads. The Intel® Pentium® 4 processor with Hyper-Threading Technology (HT) [24] is one such design. In an HT machine, each hardware thread context is exposed to the operating system (OS) as a distinct logical processor, to which the OS is responsible for binding an OS-visible thread for scheduled execution. The only way for a user program to access multiple hardware contexts in a single HT processor is to use the OS thread APIs to create threads and manage inter-thread scheduling affinity and synchronization.

As reported in a recent study [19], helper threading can achieve significant speedups when applied to physical systems equipped with HT processors. That study sheds valuable insights into several key tradeoffs. First, because some processor structures are either shared or partitioned between logical processors in the multithreading mode [24], resource contention is increased. Consequently, helper threads

must be invoked judiciously. Second, helper thread invocation should be adaptable, responding to changing program phases. For example, a particular delinquent load might experience a significant number of cache misses over the entire program execution, but the temporal distribution of those misses might not be uniform. Hence, some form of dynamic self-throttling is essential. Such a throttling method ensures that the helper threads will run neither behind nor too far ahead of the main thread. Unfortunately, due to the unpredictable and long latency associated with OS-based inter-thread synchronization, it is difficult to achieve the adequately fine-grain control necessary to tackle these issues effectively.

To address these challenges, we introduce *virtual multithreading* (VMT), which virtualizes the single instruction pointer (IP) architecture in a single logical processor environment to support multiple user-level thread contexts. The VMT mechanism is able to monitor for long-latency microarchitectural events, such as last-level cache misses, and respond by performing a fly-weight (< 100 cycles) control transfer to another code location in the same application program. Prior research [2, 13, 15, 26] has found benefit from similar switch-on-event approaches; but being geared primarily towards improving program throughput, it has typically required special OS and hardware support. In contrast, the VMT technique is focused on reducing the latency of an individual program thread. As a form of user-level switch-on-event multithreading, the VMT thread switching is OS-transparent. Once activated due to a long-latency cache miss in the main thread, the helper thread can run entirely hidden in the shadow of the main thread's outstanding cache miss, imposing no overhead on the main thread.

Leveraging existing firmware support for instruction set emulation, we prototype the VMT mechanism in a commercially available Itanium® 2 processor. To minimize the overhead for the VMT thread switch and synchronization, we also implement compiler optimizations that judiciously partition the large register sets of the Itanium architecture [18] between the main and helper threads. In this paper, we demonstrate that significant performance speedups can be achieved for a set of real-world workloads when they are optimized with VMT-based helper threading technology and run on a 4-way MP system equipped with the VMT prototype processors.

The remainder of this paper is organized as follows. Section 2 describes the VMT architecture and its usage model for supporting prefetching helper threads. Section 3 highlights our experiences in developing the firmware-based VMT prototype in the Itanium 2 processor. Section 4 presents workload performance evaluation. Section 5 reviews the related work, and Section 6 concludes.

2. VIRTUAL MULTITHREADING

In this section, we describe the virtual multithreading architecture for low overhead thread switching among event-driven user-level threads. In addition, we present a helper threading approach as an example usage model of VMT.

2.1 VMT Architecture

2.1.1 Event-driven User-level Threads

In traditional multithreading application program development, standard OS thread APIs, e.g. Windows Thread APIs, are used to create OS kernel-managed threads (OS threads for short). These OS thread APIs are also used to manage inter-thread synchronization, e.g. `SetEvent()` and `WaitForSingleObject()`, and special thread scheduling requirements such as affinity and priority. It is the responsibility of the OS to both make thread scheduling decisions (e.g. when to context-switch threads) and perform the inter-thread synchronization. The latency incurred on an OS thread context switch or synchronization operation is usually in the range of microseconds, a glacial time scale relative to the latency of a typical microarchitectural event of tens of nanoseconds. To overcome the heavy overhead for OS thread switching and synchronization, some runtime systems implement light-weight user-level thread libraries that allow the user to create OS-transparent user-level threads and to explicitly manage thread scheduling and synchronous switching without invoking the OS kernel. However, a conventional user-level thread library, such as the fiber utility [4] in the Windows OS, is unable to perform event-driven preemptive scheduling. In addition to the ability to synchronously switch between threads, VMT provides applications with the new capability to directly observe for and react to microarchitectural events on a logical processor without any OS involvement. To the OS, the switching of user-level threads is completely transparent; that is, the OS has the illusion of a single OS thread for the application program running on the processor.

One example microarchitectural event of interest is last-level cache misses that cause a pipeline stall. In response to such events, VMT enables a quick thread switch to suspend the main thread, execute a helper thread, and resume main thread execution completely within the shadow of the memory stall. Section 2.2 provides further detail into this usage model. To execute helper threads within the time to access memory, it is essential to ensure very low overhead for the VMT thread switching. We use the term *fly-weight* to signify that the control transfer is of microarchitectural time scale and to distinguish it from *light-weight* OS control transfers, which can be thousands of cycles or more.

State Swapped	Cycles	Cumulative Cycles
IP (minimum)	21	21
UNAT and GR1 – 7	36	57
GR8 – 15	12	69
GR16 – 31 (banked)	66	135
PR and LC	14	149
BR0 – 7	53	202
RSE	65	267
FR2 – 31	59	326

Table 1: Latency to perform a VMT-based thread context switch as the amount of swapped state is increased.

2.1.2 Fly-weight VMT Thread Context Switch

Table 1 shows the cycle cost for VMT-based context switch overhead as different subsets of register state are saved and restored for an Itanium 2 processor. A minimal context switch involves swapping thread IPs, ignoring all other register state. Such a context switch requires about 21 cycles. However, as additional application register state is saved and restored, the cost of a context switch increases significantly. For processors with memory latency of 200 to 300 cycles, a context switch time of 300+ cycles would

make it impossible to even complete a context switch in the shadow of a last-level cache miss. Therefore, to ensure effectiveness for a usage model like helper threading, it is essential to keep VMT context switch cost at a minimum.

To this end, a state-of-the-art compiler optimization is introduced to perform judicious register partitioning among user-level threads. By taking advantage of the large register sets of the Itanium architecture, the compiler is able to allocate a distinct subset of registers to be used by the helper threads. Therefore, it suffices to swap only IPs during a VMT thread switch.

An additional benefit of the register partitioning optimization is to reduce the cost of value communication between user-level threads. In the helper threading usage model, such synchronization manifests as the communication of live-in values from the main thread to the helper thread. Albeit explicitly partitioned, the registers used by the main thread and the helper thread belong to the same application register file, and thus are mutually accessible by each thread. Consequently, value synchronization simply involves copying register state between the main thread and the helper thread, instead of going through memory-based shared variables. For this usage model, both the main and prefetching helper threads are considered VMT threads, and referred to as such throughout this paper.

Finally, the compiler-guided encapsulation of VMT thread contexts as part of the application register state provides a third essential benefit. Since all application register state is saved and restored across OS context switches, it is *virtualized* by the OS. In a multiprocessor environment, such as a traditional SMP system, the virtualization ensures that the execution of an OS thread which had been previously suspended on one processor can be fully resumed on the same or another processor. Because VMT threads only use existing architectural register state, the OS simply context-switches the VMT-enhanced application thread just as if it were an ordinary thread. This guarantees that VMT user-level threads can be safely used in a multiprogramming and multiprocessing environment.

Previous research [19, 20, 22, 30] has demonstrated various compiler optimizations capable of automatically analyzing delinquent loads from a program and generating effective data-prefetching helper threads. The register partitioning optimization geared towards achieving minimal VMT thread switch overhead is yet another enhancement to such optimizing compiler infrastructure development.

2.2 Helper Threading Usage Model

In previously proposed helper threading schemes which assume multithreaded processors such as SMT or HT, a helper thread executes in one of the logical processors along with the main thread running *simultaneously* in another logical processor. However, for processors that do not provide such multithreading hardware features, the main thread and the helper thread cannot be executed simultaneously. With VMT, a conventional uniprocessor can be equipped with a form of fly-weight user-level switch-on-event multithreading, where thread switches are triggered upon occurrence of a long-latency cache miss. Running a helper thread in the shadow of a main thread stall due to cache miss improves main thread performance by utilizing the otherwise idle processor resources to perform effective prefetching.

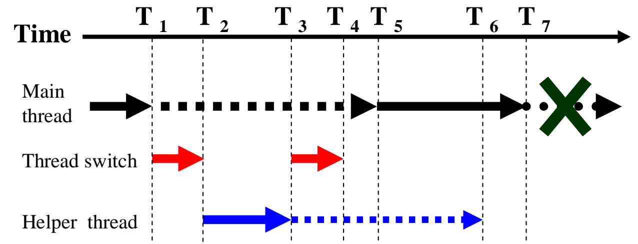

Figure 1: Prefetching Helper Thread via VMT

2.2.1 *Prefetching under a Long-latency Cache Miss*

If a processor executes a consumer instruction of a load that incurs a long-latency cache miss, the main thread is stalled waiting for the load data to be retrieved. This stall time represents an ideal opportunity to switch to a helper thread and perform data prefetching for anticipated future main thread cache misses.

Figure 1 depicts a typical scenario to execute a helper thread in the shadow of a last-level cache miss via the VMT mechanism. A main thread (solid line) encounters a long-latency cache miss at T_1. As the main thread is stalled (dotted line), the processor dynamically detects the cache miss and triggers a fly-weight thread switch to suspend the main thread and transfer control to the helper thread. At T_2, the helper thread starts the precomputation work leading to a prefetch for an address that the main thread will access and would normally miss at T_7. At T_3, the helper thread issues a prefetching load that will incur a last-level cache miss, which in turn triggers a second fly-weight thread switch, causing the helper thread to be suspended and the main thread to be resumed at T_4. The main thread at this point still waits for the original cache miss to be resolved at T_5 when the data it requested at T_1 returns from memory. Afterwards, the main thread proceeds with normal execution, which is overlapped with the cache miss initiated by the helper thread. At T_6, the data which had been requested by the prefetching helper thread at T_3 arrives in the cache. At T_7, instead of encountering yet another cache miss, the main thread observes a cache hit. In summary, this example shows that useful and effective helper thread prefetching can be performed while the main thread is stalled (i.e. time between T_1 and T_5).

For modern processors, the latency to access main memory can exceed 200 cycles. The amount of work that can be done by the helper thread per activation is given by this latency minus the overhead of VMT thread switch and synchronization. As described in Section 2.1.2, this overhead can be, in fact, quite minimal when utilizing the register partitioning optimization. As the gap between processor speed and memory latency continues to widen, the last-level cache miss time will scale up accordingly, increasing the window of opportunity for using helper threads. For example, through compiler optimization, either more helper threads can be used to target more delinquent loads, or more sophisticated helper threads can be constructed.

2.2.2 *Instructions for VMT Thread Yield*

In order for the user program to communicate to the hardware when a thread switch should occur, the VMT architecture provides two yield instructions: *Yield* and *Yield-*

Conditional. While differing in their respective triggering conditions, the Yield and Yield-Conditional instructions share the common semantics of performing a minimum context switch (i.e. swap of IPs) followed by control transfer to a VMT thread. The Yield instruction, when executed, unconditionally suspends the current thread and causes a synchronous control transfer to the VMT thread. The control transfer behaves like a mispredicted branch that would flush the pipeline.

The Yield-Conditional instruction causes a context switch only when the pipeline is stalled due to a last-level cache miss. Unlike the Yield instruction which triggers an immediate control transfer, execution is allowed to proceed past the Yield-Conditional instruction, permitting further instructions to retire. A control transfer is later invoked asynchronously if a pipeline stall due to last-level cache miss is detected. If such a stall is detected, this context switch occurs at some instruction following past the Yield-Conditional. If no stall is detected, the processor does not interrupt the current execution and the Yield-Conditional instruction behaves just like a no-op.

2.2.3 Example

Figure 2 illustrates how the yield instructions can be used in the actual VMT thread code. In the main thread on the left, the first load is a delinquent load with destination register `r3`. Note that a Yield-Conditional is placed between the load and its use, namely the `add` instruction that consumes `r3`. The Yield-Conditional instruction is responsible to check for a pipeline stall due to a last-level cache miss. If the delinquent load incurs such a long-latency cache miss, the processor will quickly stall upon the use of `r3`, that is, at the `add` instruction, while the load request is being serviced from memory. Upon detecting the triggering condition, the processor causes a VMT thread switch, followed by a control transfer to the helper thread.

As shown in Figure 2, when the helper thread is invoked for the first time, the VMT thread switch mechanism (transition 1) will transfer control to the beginning of the helper thread (transition 2a). Assuming `r6` is the live-in, the helper thread then starts to chase pointers and perform prefetching for future accesses. A helper thread can use the Yield-Conditional instructions like the main thread to trigger an event-driven VMT thread switch back to the main thread in response to a long-latency cache miss. In other words, if the prefetching load misses in the last-level cache, the helper thread will stall at its use instruction, cause a VMT thread switch (transition 3), and then transfer control back to the main thread (transition 4). If no miss is detected, the helper thread continues executing and the Yield-Conditional instruction has no side effect. Alternatively, a helper thread can use the Yield instructions to unconditionally return control to the main thread.

2.3 Key Helper Threading Characteristics

In general, a judiciously constructed helper thread uses a combination of Yield and Yield-Conditional instructions to implement a form of self-throttling to ensure the helper thread runs neither behind nor too far ahead of the main thread. To do so, both main and helper threads maintain progress counters to explicitly track their progress in the targeted program region. For example, in a loop, the corresponding counter tracks the loop iteration number being

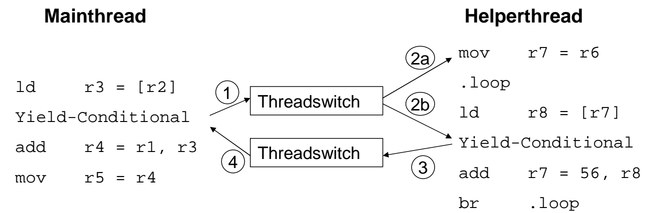

Figure 2: VMT Helper Thread Code Example

executed in either the main or helper thread. When the helper thread falls behind, that is, the progress counter of the helper thread is smaller than that of the main thread, the most recent copy of all helper thread live-in values is copied from the main thread, and its progress counter synchronized with the main thread counter value. When the helper thread runs too far ahead, the helper explicitly relinquishes control back to the main thread to allow it to catch up. In addition, a different form of self-throttling helps prevent a particular instance of a prefetching helper thread from running too long and delaying the resumption of the main thread until after its outstanding cache miss has been resolved. For instance, before the unconditional backward branch (i.e. `br .loop`) at the end of the helper thread in Figure 2, the compiler can insert self-throttling code to test if the helper thread has executed more than some maximum number of loop iterations, and to return to the main thread via a Yield if it has.

After the helper thread yields, the main thread resumes its execution as if no helper thread invocation had occurred. That is, it restarts fetch at the instruction that had triggered the helper, retaining all of its live register state as it had been prior to the thread invocation. Similarly, the VMT mechanism preserves the thread continuation for the helper threads; thus subsequent invocations of the helper resume execution at the location where the helper thread previously left from (indicated by transition 2b) instead of jumping to the beginning of the helper thread again. With continuation preserved, the main thread and the helper thread would run independently and concurrently as two *co-routines*, albeit not simultaneously. This would allow the helper thread, equipped with proper self-throttling, to maintain adequate distance ahead of the main thread.

In order to support the co-routine execution mode, for each targeted region (i.e. loop or function body), it is necessary to maintain both the initial instruction address of the beginning of the helper thread as well as the continuation instruction address which will be executed when the main thread next yields control. Thus the compiler reserves two pre-determined registers to hold those two addresses before the region is entered. During a control transfer, the thread switch mechanism reads these registers and jumps to the appropriate instruction address. By reprogramming the initial and continuation addresses when entering each different region, multiple helper threads can be supported in a single binary, albeit targeting each program region with only one helper thread. More advanced implementations are also possible that target each region with multiple helper threads, but such implementations are beyond the scope of this paper.

3. VMT PROTOTYPE ON ITANIUM 2 PROCESSOR

As a form of a user-level fly-weight switch-on-event multithreading mechanism, VMT entails basic processor support for monitoring certain microarchitectural events of interest, and responding to a detected event occurrence with a fast control transfer after a minimal context switch. In this section, we describe a novel emulation-based approach to provide such processor support for VMT on the existing Itanium 2 processor [28], which does not have any dedicated hardware support for multithreading. This approach extensively leverages the firmware mechanism [17] and the performance monitoring infrastructure [18] as building blocks.

As depicted in Figure 3, Processor Abstraction Layer (PAL) [18, 35] is a layer of firmware infrastructure. Along with System Abstraction Layer (SAL) and Extensible Firmware Interface (EFI), PAL in its traditional role maintains a consistent processor interface to the OS across multiple implementations of the Itanium Processor Family.

Executing at the kernel privilege level (also called Ring-0), the PAL firmware essentially consists of two components: a set of service procedures which provide status and control of processor capabilities and are exposed to the OS, and an ensemble of OS-transparent hardware event handlers, which are tailored to processor hardware events. It is the latter that is of particular interest in this work.

In the Itanium 2 processor, there is programmable debugging hardware support for the PAL firmware to observe and react to a wide variety of hardware events. Historically, the debugging hardware is introduced to enable comprehensive silicon debugging and validation. For instance, the PAL firmware can program the debugging hardware logic to monitor hardware breakpoint events triggered from instructions in-flight by matching the data address, instruction address, or opcode of interest. In addition, the Itanium architecture defines a set of performance monitoring unit (PMU) interfaces to track the occurrences of numerous microarchitectural events. The PAL firmware can program the PMU to count certain microarchitectural events and associate the respective counter overflow condition with a hardware breakpoint event. In turn, the debugging hardware can trigger execution of a PAL handler when the monitored PMU event occurs. Through PAL, the handling of breakpoint events is performed entirely transparently to the OS.

In the prototype system, we take full advantage of the PMU mechanism and the PAL firmware infrastructure, including the programmable debugging hardware logic, to emulate the VMT mechanism. The capability of opcode monitoring is used to emulate the two VMT yield instructions as special opcodes of interest. To monitor events that can trigger a VMT thread switch, the PAL firmware programs the PMU to track the last-level cache miss event and uses the debugging hardware logic to detect pipeline stall conditions as well as in-flight instructions with special opcodes. Upon detecting a thread yield event, the debugging hardware *directly* invokes a custom PAL handler, which is responsible for performing a minimal VMT thread switch as described in Section 2.1.2. The latency between detection of an event occurrence to the start of the custom PAL handler is equal to the cost of a pipeline flush, as in the case of a branch mispredict, plus additional overhead associated with the manipulation of certain internal processor registers.

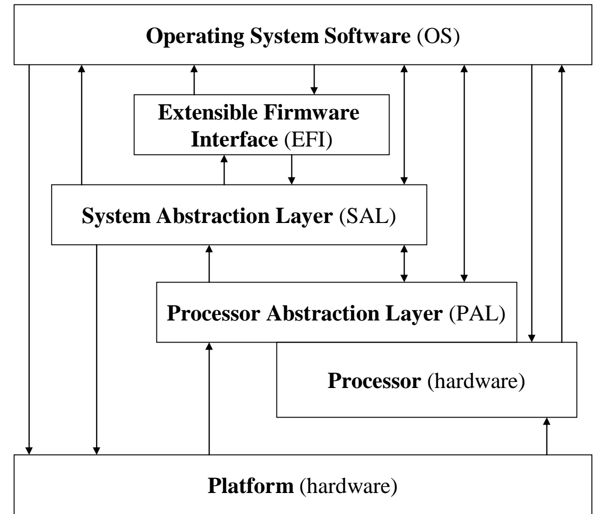

Figure 3: Itanium Processor Firmware Model

The custom PAL handler is responsible for suspending the execution of the current VMT thread, saving its continuation (at minimum, the IP), restoring the minimal context for another VMT thread, and then transferring program control to its continuation. From there on, the execution of the restored VMT thread is resumed.

On the prototyped Itanium 2 processor running at 1.5GHz, with the minimum VMT thread switch, namely, swapping only IPs between the VMT threads, the total latency between the time when a triggering event is detected and the time when control is transferred to the next VMT thread is about 70 cycles. That is, in Figure 1, both (T_2 – T_1) and (T_4 – T_3) would have a duration of about 70 cycles. For a memory latency of 200 cycles, which corresponds to the duration of (T_5 – T_1), the window of opportunity to run a helper thread, namely, the duration of (T_3 – T_2), is about 60 cycles. Obviously, with additional hardware support, the thread switch overhead can be further reduced. For example, if the overhead associated with internal processor register manipulation can be eliminated through hardware optimization, the cost of thread switch can be cut by half to as little as 35 cycles (i.e. pipeline flush plus the minimum context switch cost in Table 1). Doing so significantly expands the window of opportunity (T_3 – T_2) to 130 cycles. It is important to note that for a given processor microarchitecture, the cost of a thread switch (T_2 – T_1) is fixed, but the window of opportunity scales proportionally with the memory latency.

In the rest of this section, we share some highlights of our experience in emulating the VMT mechanism on the Itanium 2 processor.

3.1 Encoding and Trapping Yield Instructions

For the two yield instructions introduced in Section 2.2.2, the prototype system must uniquely decode both instructions and execute them in conformance to their respective architected yield semantics. It is in general impossible to introduce new opcodes on existing processor silicon without making special hardware changes. However, the PMU mechanism in the Itanium architecture provides a utility called *opcode match registers*, which allows the PAL to pro-

gram the debug hardware to recognize and trap any special opcode encoding amid the in-flight instructions. When an in-flight instruction is found to match the opcode value specified in the opcode match registers, it is tagged. As it moves downstream in the pipeline, the tagged instruction is further examined on whether it matches any back-end events. At the exception detection stage, immediately preceding retirement, the tagged instruction would trigger a hardware breakpoint event and transfer control directly to a corresponding PAL event handler, which in turn can emulate the instruction semantics. All such control transfers occur at the boundary of an *instruction*, rather than that of a bundle [18]. That is, equivalent to a user-level interrupt, a yield instruction can trigger a control transfer from or to any instruction, and is not limited according to traditional branch semantics.

Without loss of general applicability, we choose two no-op instruction encodings to uniquely represent the Yield and Yield-Conditional instructions in our prototype system. In the Itanium architecture, the no-op instruction is defined with a 21-bit immediate field, which can be potentially used by application software as an annotation marker. By default, most production compilers including those from Microsoft, Hewlett-Packard, GNU, and Intel only generate no-op instructions with a literal value of 0. We reserve two separate no-op literal values for the Yield and Yield-Conditional instructions, and modify our compiler to generate these special no-ops. The PMU opcode matching registers are also programmed to trap on these two special no-op instructions accordingly.

Using no-ops to encode the yield instructions provides an additional advantage with regard to backward compatibility; for those processors that do not support the VMT prototype mechanism, the special no-ops are simply handled as ordinary no-ops. In other words, when a binary runs on processors without the VMT prototype firmware, the yield instructions will be correctly decoded and executed as instructions with no side effect on the architectural state, and no helper threads will be invoked at runtime.

3.2 Emulating VMT Thread Yields

While the triggering condition for the Yield instruction is the detection of its corresponding no-op, the semantics of the Yield-Conditional instruction entails detection of a *conjunction* of multiple hardware events: namely, the presence of in-flight Yield-Conditional no-op, processor pipeline stall due to dependency upon a cache-missing load, and an occurrence of outstanding last-level cache miss. It is relatively easy to observe these individual events in isolation. However, in order to recognize a logical combination of these events, more sophisticated features of the PAL debugging hardware and the PMU need to be employed.

To observe and react to hardware events, the PAL debugging hardware provides a set of flexible event monitors, each of which can be programmed to detect individual microarchitectural events. The activation of one monitoring unit can be further configured to enable or disable another monitoring unit. This programmable cascading mechanism makes it feasible to use an ensemble of individual hardware events to synthesize a sophisticated composite event, such as the conjunction of triggering conditions for the Yield-Conditional. The complex composite event monitoring is entirely undertaken within PAL debugging hardware and off the critical path of the execution core pipeline. Upon detecting any such event occurrence, the debugging hardware will raise a debug breakpoint event. This in turn activates a corresponding PAL event handler that implements the minimum VMT thread context switch.

4. PERFORMANCE EVALUATION

To gauge the performance impacts of the VMT helper threads and evaluate the relevant tradeoffs, we have applied VMT-based helper threading optimization to a number of real workloads for which long-latency cache misses are known to play a significant role in their performance. Even on the emulation-based VMT prototype system, significant performance boosts can be achieved from the VMT helper threads for these workloads. In the rest of this section, we further describe the configuration of the prototype system, the characteristics of the workloads, and an in-depth analysis of the performance improvements. All performance measurements and comparisons are based on the absolute wall-clock time of program execution on the physical system.

4.1 Experimental Machine Configuration

The prototype system is a 4-way 1.5 GHz Itanium 2 processor-based MP system with 16GB of RAM. The VMT feature prototyped in the Itanium 2 processor silicon can be enabled or disabled through a special internal toggle register. Each processor's on-chip cache hierarchy consists of separate 16KB 4-way set-associative L1 instruction and data caches, a shared 256KB 8-way set-associative L2 cache, and a shared 6MB 24-way set-associative L3 cache. Through a silicon debugging mechanism, the L3 cache can be reconfigured to operate as a 1MB 4-way set-associative cache. Further details about the processor organization and platform configuration can be found in [28].

4.2 Workloads

In this research, a total of nine workloads are studied. All of them are known to suffer significantly from a large number of last-level cache misses. For workstation workloads, we choose three benchmarks: (1) MCF with reference input, (2) VPR with reference input for routing, both from the SPEC CINT2000 suite [37], and (3) DOT, a graph layout optimization tool from AT&T's Graphviz productivity suite [14]. They are all single-threaded applications.

For server workloads, we select a decision support system (DSS) running on a large IBM DB2 database [16, 31]. The DSS models a business environment that is divided into business operation support area and business decision support area. The business manages, sells, or distributes products worldwide. We choose six representative DSS queries that have long run times and span large portions of the database. These DSS queries are executed on a 100GB IBM DB2 database. This database configuration represents a sizable business environment consisting of up to 1M suppliers, 20M parts, 15M customers, 600M orders, and 25 nations. A summary of these queries is given in Table 2.

Note that these queries by themselves are *not* Itanium processor binaries; instead, they are written in SQL and processed by the same DB2 program running on the Itanium processors. Because these queries differ significantly in functionality and stress different aspects of the database, we distinguish each query as a separate benchmark.

Query	Description
QA	Retrieve 10 unshipped orders with the highest value
QB	List the revenue volume done through local suppliers
QC	Determine the value of goods shipped between certain nations to help re-negotiation of shipping contracts
QD	Determine impacts of less expensive modes of shipping on critical-priority orders
QE	Rank the top 100 customers based on their having placed a large quantity of orders
QF	Identify certain suppliers who were not able to ship required parts in a timely manner

Table 2: Description of DSS Queries

The DB2 database used in our experiment has been highly tuned to sustain greater than 95% CPU utilization for all six queries executed on the prototype system. It is important to note that the IBM DB2 program has been highly optimized for task-level parallelism. In particular, OS-visible threads are extensively used to overlap long-latency disk I/O operations with a myriad of CPU-intensive data crunching tasks. For example, during a typical DSS query processing on the 4-way MP prototype system, each physical processor has over 40 OS-threads running concurrently. Thus the wall-clock time performance measured for the DSS queries represents the overall throughput of DB2's threads. In comparison, the wall-clock time performance for the three workstation benchmarks represent the latency of a single OS-visible thread.

Since the VMT mechanism is agnostic to the OS, the workloads can run on either Microsoft Windows or Linux. We demonstrate this by running DOT on RedHat Linux Advanced Server 3.0, and MCF, VPR, and DSS on Windows Server 2003. For all four programs, the baseline binaries are built using the Intel Electron compiler [5, 21] with the best applicable optimizations. For instance, in addition to the standard –O3 optimization, both MCF and VPR are further optimized using multi-file inter-procedural optimization, profile-guided optimization, and optimizations specific to Itanium 2 processor. We next briefly highlight the baseline binary characteristics and three helper thread optimizations that have been applied in combination to the programs studied in our experiment.

4.2.1 Baseline Binary Characteristics

To build the VMT-optimized binary for a program, we use a research VMT optimizing compiler, which is based on the Intel Electron compiler, to recompile a subset of the program files that contain the delinquent loads and automatically generate the VMT helper threads as part of the object file. The resulting object files are then linked with other original object files to produce the new VMT-optimized binary. Besides the generic helper thread optimizations [19], the compiler implements several VMT-specific optimizations, including register partitioning to minimize both the VMT thread synchronization overhead and the VMT thread context switch overhead, as described in Section 2.1.2. A more comprehensive account of the enhancements to the Electron compiler is beyond the scope of this paper.

In addition, the compiler incorporates aggressive software prefetching [27] based on profile-guided feedback. Where applied, this software prefetching is highly effective. However, there remains a significant number of delinquent loads that are difficult to target with traditional software prefetching techniques for two primary reasons.

First, several targeted loads involve pointer-chasing with dependent memory operations. Such memory accesses are difficult to prefetch using software-based techniques because there is a significant risk of inducing stalls into the program. When prefetching a dependent sequence of memory operations, a cache miss for an intermediate load will cause a stall when the software prefetching code attempts to use this result. Because our helper threads utilize a Yield-Conditional instruction between each pair of dependent loads, they do not impose a risk of stalling the processor, but instead immediately yield control back to the main thread in the event of an off-chip memory access.

Second, some delinquent loads involve computationally expensive address computations. Targeting such loads imposes execution overhead when these costly instruction sequences cannot be hidden by aggressive instruction scheduling. Our VMT-based approach, however, only invokes helper threads when the main thread would be otherwise stalled, effectively hiding these expensive address computations without risk of slowing down normal main thread progress.

4.2.2 Co-routine Execution Mode

As discussed in the example given by Figure 2, the register state of suspended VMT threads is maintained in the register file across VMT thread invocations. Thus, when the main thread next incurs a long-latency cache miss, the helper thread can resume execution at its last saved thread continuation point, rather than always restarting execution from the beginning of the thread. Helper threads utilizing this co-routine execution mode can execute aggressively ahead of the main thread. For example, consider a loop in which instructions necessary to compute the loop-control [42] incur a limited number of cache misses, whereas the load instructions in the loop-body incur a significant number of cache misses. In this case, the VMT optimizing compiler constructs a helper thread that captures the loop-control computation and issues a software prefetch (i.e. `lfetch`) to target the delinquent loads in the loop-body. For the main thread, the compiler places an Yield-Conditional next to the loop-body delinquent load such that a miss on the delinquent load will be turned into a trigger that activates the helper thread. As the helper thread progress is only limited by the load in the loop-control, it can potentially execute multiple loop iterations ahead of the main thread. In this co-routine execution mode, when the main thread again incurs a cache miss, the helper thread may resume execution still several loop iterations ahead of the main thread's position in the loop, enabling continued prefetching of future loads far ahead of the main program.

4.2.3 Adaptive Wavefront Prefetching

VMT enables long-range prefetching of delinquent loads that can run far ahead of the main thread. For code where the result of a delinquent load is consumed immediately by a branch, it may be impossible to accurately predict or compute control flow within the helper thread without causing a stall. In such cases, we found it highly profitable to combine and overlap the simultaneous execution of multiple control-flow paths. For example, when prefetching a binary tree, the choice of which node to visit next may be

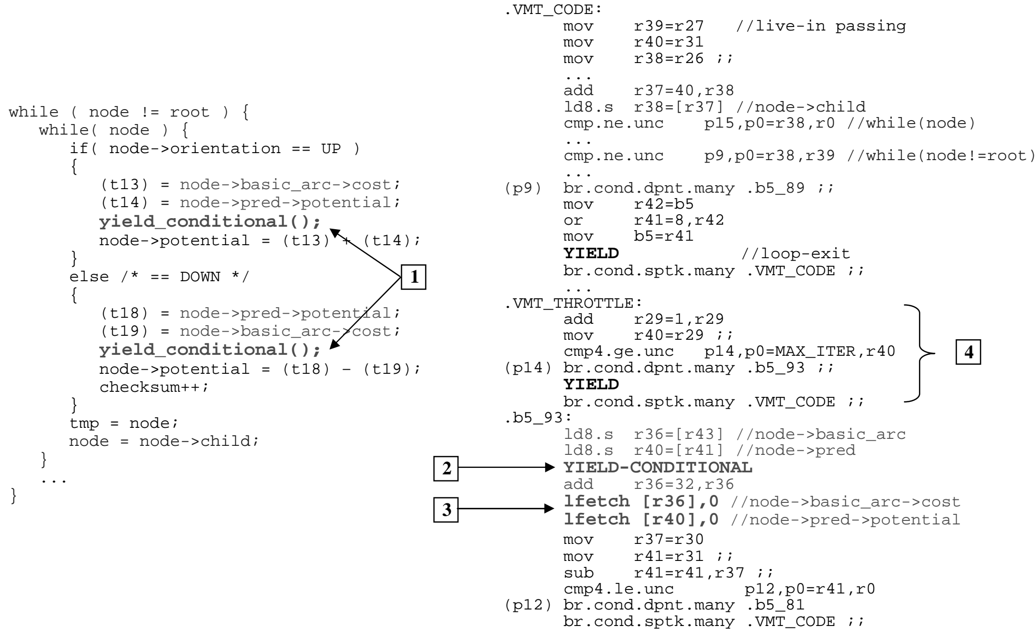

Figure 4: MCF code example - `refresh_potential()`

dictated by the result of a delinquent load. We address this problem by simultaneously visiting both the left and right children, an approach with some similarity to *wavefront prefetching* [33]. However, the self-throttling logic in the helper thread frequently checks the true control path taken in the main thread, and judiciously prunes the portion of the wavefront that runs astray from the main thread. This adaptation of the prefetching wavefront further ensures the efficiency of VMT helper threads.

4.2.4 Aggressive Use of Itanium ISA Features

Since a helper thread, by design, does not contribute to program semantics [10, 22, 41, 43], it is much more amenable to aggressive speculation than the main thread. We found a number of features in the Itanium 2 processor, which permit the control of speculative execution, to be highly useful in constructing aggressive helper threads. For instance, predication removes control-flow dependences among instructions, reducing the number of branch predictions, and hence, branch mispredictions, which can impact a helper thread that encapsulates complex data-dependent control flow. Use of branch hints allows us to further reduce the number of incurred branch mispredictions. Finally, the use of speculative and non-faulting load instructions (`ld.s`) enables the construction of aggressive helper threads that can tolerate dereferencing possibly invalid pointers.

4.2.5 Example: MCF

Figure 4 demonstrates a code example for the targeted loop in `refresh_potential()` of MCF where the main thread source code is shown on the left and the helper thread assembly code is on the right. In the main thread, Yield-Conditional instructions, labeled 1, have been added right before the use of the targeted delinquent loads in the loop-body. In the helper thread, a Yield-Conditional has been added after the first-level pointer dereference, labeled 2, which is a blocking load that serializes the execution of the delinquent loads. However, since the targeted delinquent loads themselves are in the loop-body and their loaded values are not used to compute either the loop-control or effective load addresses, they can be safely converted to `lfetch`, labeled 3, which is a non-blocking prefetch instruction supported in the Itanium architecture. Label 4 is an example of the self-throttling code in MCF.

4.3 Speedups and Analysis

For each workload, when the VMT feature is disabled in the prototype system, the baseline binary and the VMT-optimized binary have the same performance. This is because yield instructions behave like true no-ops when VMT is disabled, and no helper threads will be invoked. Additionally, when compiled without VMT support, the studied benchmarks achieved identical performance whether the VMT feature was enabled on the processor or not. This is also expected since the VMT firmware feature is implemented in dedicated PAL and PMU hardware and off the critical path in the processor core. For the rest of this section, by performance comparison between a baseline binary and its VMT-optimized counterpart, we always mean that the comparison is carried out on the prototype system with the VMT firmware feature *enabled*.

Figure 5 shows that significant performance speedups can be achieved from applying VMT prefetching helper threads for these memory-intensive workloads, ranging from 5.0% in DSS query QD to 38.5% in MCF.

To further shed insights into the effectiveness of the VMT helper threads, Table 3 provides details on the static and dynamic characteristics of the VMT helper threads and their targeted delinquent loads. For each benchmark, the table reports the number of delinquent loads targeted (*Targeted loads*), the number of helper threads constructed (*Helper threads*), the size of the helper threads in terms of instructions (*Helper instrs*), the number of stop-bits per helper thread (*Helper stop-bits*), the percentage of total execution

Benchmarks		*Targeted loads*	*Helper threads*	*Helper instrs*	*Helper stop-bits*	*Mem stall time*	*Miss reduction*	*Miss reduced per switch*
DOT		4	1	32	9	26.7%	21.7%	0.3
MCF		5	1	50	25	44.2%	83.7%	5.1
VPR		4	1	35	26	27.3%	47.9%	2.7
DSS	QA	4	2	90 / 28	61 / 19	29.9%	53.2%	1.1
	QB					20.3%	51.2%	1.3
	QC					25.7%	52.8%	1.1
	QD					10.7%	52.1%	1.1
	QE					38.9%	19.6%	0.2
	QF					24.5%	57.4%	1.3
Arithmetic mean:						27.6%	48.8%	1.6

Table 3: Vital Characteristics of VMT Helper Threads

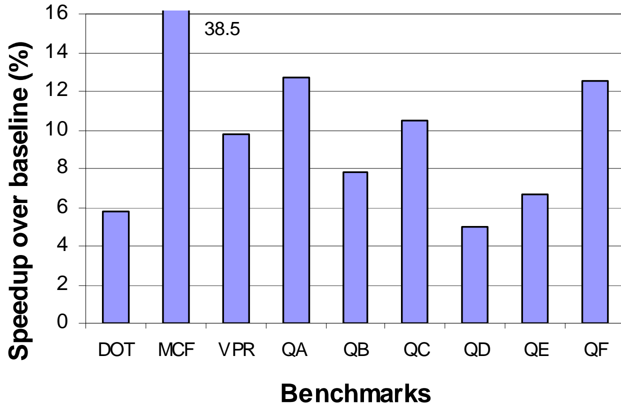

Figure 5: Speedups of VMT Helper Threads

time due to memory stalls by the targeted delinquent loads (*Mem stall time*), the fraction of the baseline's L3 cache misses that are eliminated after running helper threads (*Miss reduction*), and the average number of L3 misses reduced each time the a helper thread is switched (*Miss reduced per switch*). Because the Itanium Processor Family requires that data dependences be explicitly delimited using stop bits, the number of stop bits in a helper thread indicate the helper thread critical path length. From this data we can draw several important observations.

First, even for such a large scale workload as DSS, simply by using one or two VMT helper threads to target a handful of delinquent loads, we can achieve significant speedups for a variety of complex queries. Such efficacy is indicative of both the criticality of the targeted delinquent loads, and the quality of the helper threads. As *Miss reduction* data in Table 3 shows, a significant fraction of the L3 cache misses, 48.8% on average for the nine benchmarks, are offloaded from the main thread's critical path onto the helper thread.

Second, the size of each helper thread in terms of the number of instructions and stop-bits clearly reflects the nontrivial amount of computations that are entailed to accurately prefetch for targeted delinquent loads in a timely manner. Further analysis of these delinquent loads reveals that these loads tend to be performing sophisticated pointer-chasing traversals of complex dynamic data structures, such as BTrees or hash tables, and incurring significant amount of capacity misses at the last-level cache, despite its capacity of 6MB. The excessive number of capacity misses can be attributed to the inherent nature of the foundational database operations like sorting and hash-join, which span the entire database, and by design, exhibit neither locality nor regular patterns of consecutive memory accesses. Therefore, it is not surprising that traditional pattern-based hardware prefetchers or stride-based software prefetching schemes are often of limited effectiveness for such pointer-intensive workloads.

Third, as a unique user-level thread decoupled from the main thread, a helper thread can encapsulate complex control-flow dependencies in addition to data-flow slices extracted from the main thread. As explained in Section 4.2, very aggressive helper threads can be constructed by the VMT optimizing compiler. In fact, since helper threads are speculative, they can incorporate highly aggressive optimizations, such as described in Section 4.2.3, that could not have been safely applied by the compiler to the main thread. Consequently, helper threads, in many cases, eliminate multiple L3 cache misses for each invocation. For example, MCF eliminates 5.1 L3 cache misses per invocation, and VPR eliminates 2.7 as shown in Table 3. As the window of opportunity for VMT thread widens along the increasing gap between processor speed and memory latency, the aggressiveness and sophistication of the helper thread can scale up accordingly.

Finally, as exhibited in the performance gains for DSS workloads, even though helper threading was originally motivated as a technique to improve single thread latency [41], the throughput performance of highly threaded workloads can also be improved by reducing the latency of individual constituent threads. As demonstrated through the achieved performance gains, the functional correctness of the highly threaded VMT-enhanced DB2 binary is a powerful testimony that our VMT prototype system and compiler technology succeed in achieving the virtualization of VMT thread states in the multiprogramming and multiprocessing environment.

4.4 Scalability Study

VMT helper threads have a unique characteristic in that they are only activated upon hardware detection of long-latency last-level cache miss events, and once activated, they typically execute entirely within the shadow of the outstanding cache miss. This invocation strategy naturally throttles helper thread execution according to the number of cache misses the program experiences in the current program phase. Over the lifetime of program execution, the helper thread activities will automatically ramp up as the main thread encounters more long-latency cache misses, and automatically ramp down as effective prefetching reduces the number of experienced cache misses. As the gap of pro-

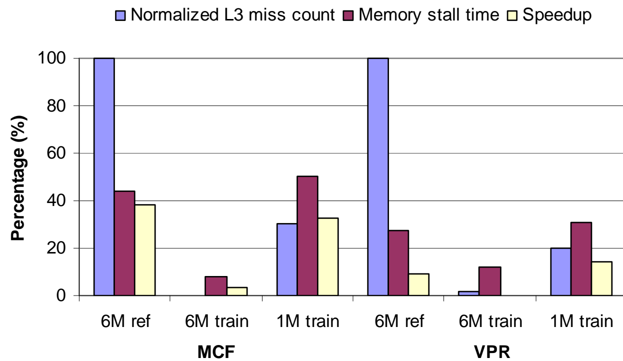

Figure 6: VMT Helper Thread Results for Various Processor Configurations

cessor speed and memory latency grows, the inherent dynamic adaptability in VMT helper threads *should* translate directly to performance scalability.

To quantify this hypothesis, we demonstrate VMT helper threads' dynamic adaptability to the program behavior across different machine configurations. To this end, we further investigate two SPEC benchmarks, MCF and VPR. In this experiment, we change each program's cache behavior by using different input sets between the reference input and the train input. In addition, we also vary the cache configuration between 6MB 24-way L3 cache and 1MB 4-way L3 cache. The helper threads we utilize are the same as explored in Section 4.3.

Figure 6 presents data as cache size and input size are varied. Results are shown for each program as three groups of three bars, with MCF's bars on the left, and VPR's on the right. For each benchmark, the three groups of bars are, from left to right, a 6MB L3 cache executing reference input, a 6MB L3 cache executing train input, and a 1MB L3 cache executing train input. Within each group of bars, we show, from left to right, the normalized L3 miss count relative to the 6MB 24-way reference input configuration, the fraction of execution time occupied by memory stalls from the targeted delinquent loads, and the speedup over a processor which does not employ VMT. All results are shown as percentages.

From 6M-ref to 6M-train, the number of L3 cache misses is significantly reduced, almost to zero compared to the 6M-ref result. This is because the data working set of the train input fits within the 6MB cache. As the L3 cache misses disappear, the portion of execution time spent on memory stalls for the targeted loads becomes smaller as well. This reduces the speedup provided by VMT helper threads accordingly.

However, these results actually indicate a strength of the VMT-based helper threading technique. Because there are insufficient cache misses for helper threads to target, by virtue of the thread spawning mechanism, helper threads are rarely spawned. Such helper threads would have performed useless work anyway, as the targeted data is already present in the L3 cache. Had these threads been executed indiscriminately, they would have caused execution overhead and slowed main thread execution [20, 43].

When the L3 cache size is reduced to 1MB, which results in even the train input incurring cache misses, performance gains from the helper threads are increased accordingly. As more cache misses occur, helper threads are spawned more frequently, and have more opportunity to provide performance gains.

These results demonstrate a key strength of this technique: the VMT helper threads are easily adapted to changing execution domains, both in terms of varying hardware platforms and changing program input sizes. Binaries compiled for one configuration could be run on multiple different configurations without concern for negative performance impacts on the program thanks to the natural throttling inherent in the technique itself.

5. RELATED WORK

The topic of helper threading has received intense research attention. Most previous work [9, 10, 23, 34, 43] has studied helper threading on top of a processor equipped with Simultaneous Multithreading, though some approaches assume a separate *helper pipeline* [3, 25]. Nearly all such studies have been simulation-based, with some exploring the impact of applying helper threading to a physical system [19, 41].

Several processors have implemented special hardware support for switch-on-event multithreading [6, 12] as a way to multiplex multiple OS level threads on a single processor core. VMT implements a user-level switch-on-event multithreading for helper threads without requiring hardware support for maintaining multiple thread contexts, and is capable of multiplexing user-level threads within the same OS thread.

Proposals have been made for run-ahead prefetching techniques [11, 29]. Under these optimizations, rather than stalling due to a long-latency cache miss, the processor continues to speculatively fetch and execute instructions for the sake of prefetching. Unlike helper threading, the run-ahead technique is fundamentally limited to speculatively executing a single stream of instructions. The effectiveness of run-ahead is highly sensitive to the predictability of the control flow and the number of data cache accesses that are independent from the cache-missing loads. In contrast, VMT-based helper threading is equipped with multiple concurrent logical sequencers, each having different control flow and together running as co-routines. For workloads like DB2, due to the nature of BTree and hash table manipulation algorithms, the control flows are usually unbiased and highly data-dependent upon the cache-missing loads. It is challenging to make run-ahead effective for such data access behaviors. However, more sophisticated helper threads, such as in adaptive wavefront prefetching as described in Section 4.2.3, can be constructed to remove control flow uncertainty and to aggressively prefetch multi-pronged data structures.

Prior to VMT, there have been various other techniques exploiting the idea of thread switching upon detecting a cache miss. However, those techniques are primarily used to increase *throughput* with multiple threads running. In contrast, the VMT-based helper threading technique is designed to reduce *latency* of a single program thread. Block multithreading [13] and differential multithreading [15] improve throughput by assuming multiple instruction streams share a single pipeline, and interleaving instructions by issuing from another thread when one thread is blocked. April [2] switches threads via the OS when one thread issues a network request to fetch from remote memory. In contrast, VMT requires no hardware support for multiple

thread contexts, and introduces a significantly faster and OS-transparent context switch technique.

The register partitioning compiler optimization for VMT also has some similarities to other previous research. Register relocation [40] is a hardware mechanism for dynamically partitioning the register usage of multiple threads. Mini-threads [32] statically partitions the registers equally among the threads. Even though our technique is also entirely compiler driven, it does not evenly partition the register sets amongst all VMT threads. Instead, the compiler honors the asymmetry in the size of register working sets used respectively by the main thread and the helper threads, and only dedicates very few registers to the helper threads.

In contrast to all previous research in the related areas, our VMT technique is the first work of its kind to explore the impact of a helper threading implementation on a physical machine supporting the Itanium architecture. Through VMT, we demonstrate the capability to multiplex the execution of multiple user-level threads on a microarchitecture without any explicit multithreading support, and without any additional OS support. The fly-weight event-driven user-level VMT thread switching mechanism prototyped in our experiment has a significantly lower thread switching overhead than previous techniques employed for user-level threads.

6. CONCLUSION

This paper makes three significant contributions. First, we introduce the concept of virtual multithreading, or VMT, which can virtualize a uniprocessor architecture to support multiple concurrent user-level thread contexts. VMT is OS-transparent and requires no explicit multithreading hardware support. The VMT mechanism is capable of monitoring pipeline stall conditions due to last-level cache misses, and respond to them by performing a fly-weight switch to another thread. Through compiler-guided register partitioning, synchronization overhead between the main thread and helper threads is kept to a minimum. For the helper threading usage model, helper threads can be activated to execute entirely within the shadow of long-latency cache misses incurred by the main thread.

Second, we demonstrate a highly productive empirical approach to VMT research. Through innovative exploitation of the silicon debugging mechanism already existing in the Itanium 2 processor, we successfully build an emulation-based VMT prototype processor without any extra special hardware. In addition, we develop a set of powerful VMT-specific compiler optimizations on top of a state-of-the-art production compiler for the Itanium architecture. With both prototype hardware and compiler support, we are able to apply VMT-based helper threading optimizations to a diverse set of real-world workloads and run the resulting binaries with functional correctness. Most significantly, even on such emulation-based prototype physical system, VMT-based helper threading can achieve significant performance improvement for these workloads. On a 4-way MP system equipped with the VMT prototype Itanium 2 processors, we measured wall-clock speedups of 5.8% to 38.5% for the workstation benchmarks, and 5.0% to 12.7% on various queries in a large scale DSS workload on the IBM DB2 Universal Database.

Third, our extensive performance evaluation of VMT-enabled workloads sheds key insights into the architectural tradeoffs unique to the VMT mechanism and the essential characteristics innate to the VMT helper threads. Being highly adaptive to the program's dynamic behavior, VMT helper threads can achieve impressive performance scalability for a variety of processor configurations. As the gap between the processor speed and memory latency continues to widen, VMT helper threads can become even more important, improving not only latency performance, but also throughput performance. Our future work will further fine-tune the compiler optimizations, investigate more efficient hardware mechanisms to enable VMT, and explore more usage models that can benefit from the VMT architecture.

7. ACKNOWLEDGEMENTS

We appreciate the useful comments on the early draft from the many referees. We would also like to thank Hank Levy for his valuable suggestions for improving the quality of this paper; and Ryan Rakvic, Natalie Enright, and Jeff Brown for further editing comments.

We thank Xinmin Tian, Kaylan Muthukumar, Gerolf Hoflehner, Dan Lavery, and Shih-wei Liao for their contribution to the compiler work described in this paper. We also thank Ashok Seshadri, Anthony Mah, and Piyush Desai who assisted us in the emulation-based prototyping work. Finally, we thank Richard Wirt, Steve Pawlowski, Bryant Bigbee, Kumar Balasubramanian, Wei Li, and Milind Girkar for their sustained support of our VMT research work.

8. REFERENCES

[1] T. Aamodt, P. Marcuello, P. Chow, P. Hammarlund, H. Wang, and J. Shen. Hardware Support for Prescient Instruction Prefetch. In *10th International Symposium on High Performance Computer Architecture*, February 2004.

[2] A. Agarwal, B. Lim, D. Kranz, and J. Kubiatowicz. April: A Processor Architecture for Multiprocessing. In *17th International Symposium on Computer Architecture*, June 1990.

[3] M. Annavaram, J. M. Patel, and E. S. Davidson. Data Prefetching by Dependence Graph Precomputation. In *28th International Symposium on Computer Architecture*, pages 52–61, Goteborg, Sweden, June 2001. ACM.

[4] D. Berg and B. Lewis. *Threads Primer: A Guide to Multithreaded Programming*. SunSoft Press, 1996.

[5] J. Bharadwaj, W. Chen, W. Chuang, G. Hoflehner, K. Menezes, K. Muthukumar, and J. Pierce. The Intel IA-64 Compiler Code Generator. *IEEE Micro*, Sept-Oct 2000.

[6] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. Kunkel. A Multithreaded PowerPC Processor for Commercial Servers. *IBM Journal of Research and Development*, 44(6):885–898, 2000.

[7] R. S. Chappell, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous Subordinate Microthreading (SSMT). In *26th International Symposium on Computer Architecture*, pages 186–195, Atlanta, GA, May 1999. ACM.

[8] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt. Difficult-path Branch Prediction Using Subordinate Microthreads. In *29th International Symposium on Computer Architecture*, Anchorage, AK, May 2002.

[9] J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic Speculative Precomputation. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 306–317, Austin, TX, December 2001. ACM.

[10] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *28th International Symposium on Computer Architecture*, July 2001.

[11] J. Dundas and T. Mudge. Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss. In *11th Supercomputing Conference*, July 1997.

[12] R. Eickemeyer, R. Johnson, S. Kunkel, B.-H. Lim, M. Squillante, and C. Wu. Evaluation of Multithreaded Processors and Thread Switch Policies. In *International Symposium on High Performance Computing*, pages 75–90, Fukuoka, Japan, November 1997.

[13] M. K. Farrens and A. R. Pleszkun. Strategies for Achieving Improved Processor Throughput. In *18th International Symposium on Computer Architecture*, May 1991.

[14] Graphviz – open source graph drawing software. http://www.research.att.com/sw/tools/graphviz/.

[15] J. W. Haskins Jr., K. R. Hirst, and K. Skadron. Inexpensive Throughput Enhancement in Small-Scale Embedded Microprocessors with Block Multithreading: Extensions, Characterization, and Tradeoffs. In *20th International Performance, Computing, and Communications Conference*, April 2001.

[16] IBM DB2 Product Family. http://www.ibm.com/software/data/db2/.

[17] *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*. Intel Corporation, June 2002.

[18] *Intel Itanium Architecture Software Developer's Manual*. Intel Corporation, October 2002.

[19] D. Kim, S. Liao, P. Wang, J. del Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. Shen. Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors. In *International Symposium on Code Generation and Optimization*, March 2004.

[20] D. Kim and D. Yeung. Design and Evaluation of Compiler Algorithms for Pre-Execution. In *10th Architectural Support for Programming Languages and Operating Systems*, pages 159–170, October 2002.

[21] R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, C. C. Lim, J. Ng, and D. Sehr. An Advanced Optimizer for the IA-64 Architecture. *IEEE Micro*, Nov-Dec 2000.

[22] S. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. Shen. Post-Pass Binary Adaptation for Software-Based Speculative Precomputation. In *ACM Conference on Programming Language Design and Implementation*, June 2002.

[23] C. K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *28th International Symposium on Computer Architecture*, June 2001.

[24] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, February 2002.

[25] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. Slice Procesors: an Implementation of Operation-based Prediction. In *International Conference on Supercomputing*, June 2001.

[26] T. C. Mowry, C. Q. Chan, and A. K. Lo. Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory. In *4th International Symposium on High Performance Computer Architecture*, February 1998.

[27] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[28] H. Muljono, S. Rusu, B. Cherkauer, and J. Stinson. New 130nm Itanium 2 Processors for 2003. In *Hot Chips*, 2003.

[29] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors. In *9th International Symposium on High Performance Computer Architecture*, January 2003.

[30] V. Panait, A. Sasturkar, and W.-F. Wong. Static Identification of Delinquent Loads. In *International Symposium on Code Generation and Optimization*, March 2004.

[31] M. Poess and C. Floyd. New TPC Benchmarks for Decision Support and Web Commerce. http://www.tpc.org.

[32] J. Redstone, S. Eggers, and H. Levy. Mini-threads: Increasing TLP on Small-Scale SMT Processors. In *9th International Symposium on High Performance Computer Architecture*, February 2003.

[33] A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linked data structures. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 1998.

[34] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. In *7th IEEE International Symposium on High Performance Computer Architecture*, Jan 2001.

[35] R. Sites. *Alpha Architecture Reference Manual*. Digital Press, Newton, MA, 1992.

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

[37] SPEC CPU2000 Documentation. http://www.spec.org/osg/cpu2000/docs/.

[38] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *22nd International Symposium on Computer Architecture*, June 1995.

[39] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor. In *5th International Symposium on High Performance Computer Architecture*, January 1999.

[40] C. A. Waldspurger and W. E. Weihl. Register Relocation: Flexible Contexts for Multithreading. In *20th International Symposium on Computer Architecture*, May 1993.

[41] H. Wang, P. Wang, R. D. Weldon, S. Ettinger, H. Saito, M. Girkar, S. Liao, and J. Shen. Speculative Precomputation: Exploring Use of Multithreading Technology for Latency. *Intel Technology Journal*, February 2002.

[42] P. Wang, H. Wang, J. Collins, E. Grochowski, R. Kling, and J. Shen. Memory latency-tolerance approaches for Itanium processors: Out-of-order Execution vs. Speculative Precomputation. In *8th International Symposium on High Performance Computer Architecture*, Feb 2002.

[43] C. Zilles and G. Sohi. Execution-based Prediction Using Speculative Slices. In *28th International Symposium on Computer Architecture*, July 2001.