Multiple Instruction Stream Processor

Richard A. Hankins Gautham N. Chinya Jamison D. Collins Perry H. Wang
Ryan Rakvic* Hong Wang John P. Shen

Microarchitecture Research Lab
Microprocessor Technology Labs, Corporate Technology Group
Intel Corporation
Contact: richard.a.hankins@intel.com

Abstract

Microprocessor design is undergoing a major paradigm shift towards multi-core designs, in anticipation that future performance gains will come from exploiting thread-level parallelism in the software. To support this trend, we present a novel processor architecture called the Multiple Instruction Stream Processing (MISP) architecture. MISP introduces the sequencer as a new category of architectural resource, and defines a canonical set of instructions to support user-level inter-sequencer signaling and asynchronous control transfer. MISP allows an application program to directly manage user-level threads without OS intervention. By supporting the classic cache-coherent shared-memory programming model, MISP does not require a radical shift in the multithreaded programming paradigm. This paper describes the design and evaluation of the MISP architecture for the IA-32 family of microprocessors. Using a research prototype MISP processor built on an IA-32-based multiprocessor system equipped with special firmware, we demonstrate the feasibility of implementing the MISP architecture. We then examine the utility of MISP by (1) assessing the key architectural tradeoffs of the MISP architecture design and (2) showing how legacy multithreaded applications can be migrated to MISP with relative ease.

1. Introduction

Across the industry, the relentless pursuit of higher clock frequency for achieving better single-thread (ILP) performance has abruptly stalled due to the untenable power scaling required to continue this approach. As a result, many microprocessor manufacturers are designing future processors with multiple processor-cores per die to improve thread-level (TLP) performance. As multi-core processors become more widely available, there will be considerable impetus towards developing multithreaded applications that take advantage of these processors. However, two significant challenges can potentially impede the progress of enabling multithreaded applications on multi-core processors.

First, tremendous effort is required to design or retool an operating system (OS) to effectively support a large number of cores, as evidenced in [5]. Contemporary OSs play a vital role in managing the execution of multithreaded applications on multi-core processors and multi-socket multiprocessing systems. Application programs use an OS-provided application programming interface (API) for thread control and thread synchronization, and threads are scheduled for execution by the OS. Unfortunately, legacy OSs may not be able to effectively deal with the increasing number of cores in multi-core processors. It is highly desirable to seek alternatives to the OS-based threading interface that allow multithreaded applications to scale to a larger number of cores independent of OS support.

Second, most multiprocessor OSs are designed to manage symmetric or homogeneous processing cores. Research has shown that asymmetric or heterogeneous cores can deliver higher performance at a given power budget than a homogeneous multi-core design [9, 15, 17, 18]. However, as Balakrishnan et al. observed [2], the asymmetry in the heterogeneous or asymmetric cores can lead to unpredictable and adverse execution behaviors if thread executions on these cores are managed by the OS.

To address these challenges, this paper presents the *Multiple Instruction Stream Processing* (MISP) architecture. MISP defines a MIMD ISA extension that introduces a new form of architectural resource, called *sequencer*, as the abstraction of a processor core. Each MISP processor consists of two or more sequencers: one sequencer is managed

*Ryan Rakvic is now an Assistant Professor with the United States Naval Academy, in Annapolis, Maryland. This research was performed while he was a member of the Microarchitecture Research Lab at Intel.

Proceedings of the 33rd International Symposium on Computer Architecture (ISCA'06)
0-7695-2608-X/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

by the OS while the rest are managed by the application program. With respect to the OS managed sequencer, the application managed sequencers behave as MIMD "functional units." Each application managed sequencer runs a distinct instruction stream as a user-level thread in parallel with other sequencers; such a user-level thread is called a *shred*. To the OS, a MISP processor appears as a single logical CPU on which OS threads can be scheduled. Each OS thread can consist of multiple shreds that can run concurrently on multiple sequencers in the MISP processor. Within a MISP processor, the sequencers may be heterogeneous or asymmetric in terms of performance, power or functionality. Since the application manages these sequencers, there is no need for the OS to cope with the asymmetry among sequencers. Furthermore since all sequencers in a MISP processor operate within the same OS thread and share the same virtual address space, the MISP architecture preserves the widely-accepted shared-memory programming model. With this essential architectural support for shared virtual address space amongst sequencers, MISP-enabled applications or runtime software are relieved of the complexity of having to explicitly manage partitioned code and data working sets between distributed memory spaces [8].

Rather than defining new instructions that directly match all the high level multithreading primitives (e.g. fork, synchronization, etc.) [16], the MISP ISA extension takes a minimalist approach by defining a canonical set of sequencer-aware instructions that support inter-sequencer signaling and asynchronous control transfer. The new instructions are sufficient for supporting the prevalent shared-memory multithreaded programming model. In fact, one can easily use the MISP ISA extension to construct a POSIX-compliant user-level multithreading runtime software infrastructure, which can hide the new instructions behind a familiar, legacy threading API, such as the POSIX Threading API (Pthreads). Consequently, the MISP multicore implementation is relieved of the complexity needed to manage shred contexts and shred scheduling in hardware.

This paper makes the following contributions:

1. We introduce MISP, a novel MIMD extension to the IA-32 ISA, as an alternative to the OS-based approach by which a multithreaded application program can directly manage multiple architectural processor-core resources. This MISP architecture and execution model can coexist with the legacy OS-based approach to multiprocessing.

2. Through the use of special firmware, we prototyped an implementation of the MISP architecture using a physical IA-32-based multiprocessor system. With this prototype system and a set of representative large-scale multithreaded applications, we examine the key architectural tradeoffs and demonstrate the feasibility to efficiently implement an IA-32 MISP processor. Beyond the MISP uniprocessor, we further extended our prototype to emulate an MP system with multiple MISP processors and report the initial evaluation of this MISP MP prototype.

3. We provide empirical evidence that the migration of legacy software from the traditional SMP targets to the MISP architecture can be accomplished with relative ease due to MISP's support for the shared-memory programming model.

The remainder of the paper is organized as follows. Section 2 introduces the MISP architecture and execution model. Section 3 presents the MISP programming model and software environment. Section 4 describes an experimental prototype system for an implementation of a MISP processor and a runtime software layer that provides POSIX-compliant threading API support to the application developer. Section 5 assesses the key MISP architectural tradeoffs through an analysis of multithreaded workloads executing on the MISP prototype system. Section 6 discusses how our work differs from previous research. Section 7 presents conclusions and indicates areas for future work.

2. MISP Processor Architecture

The MISP architecture provides a canonical set of ISA mechanisms that support execution of user-level threads on the asymmetric sequencers. For convenience, we use *shreds* to denote these MISP-enabled user-level threads, and describe an application that is parallelized with shreds as being *multi-shredded*. Figure 1 depicts a multi-shredded application running as a single OS thread on a MISP processor that consists of one *OS-managed sequencer* (OMS) and three *application-managed sequencers* (AMS). This MISP processor is presented to the OS as a single logical processor capable of running one OS thread at a time.

The remainder of this section provides further detail on the key features in an IA-32-based MISP ISA design.

2.1 Sequencer as Architectural Resource

A *sequencer* corresponds to a hardware thread context that is capable of fetching and executing one stream of instructions. Multiple sequencers in a MISP processor can be physically implemented through a variety of techniques, including simultaneous multithreading (SMT), chip-multiprocessor (CMP), or even multi-socket multiprocessors. And much like how register and memory architectural resources can be virtualized, the sequencer resource can be virtualized as well; the mechanism to map virtual

0-7695-2608-X/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

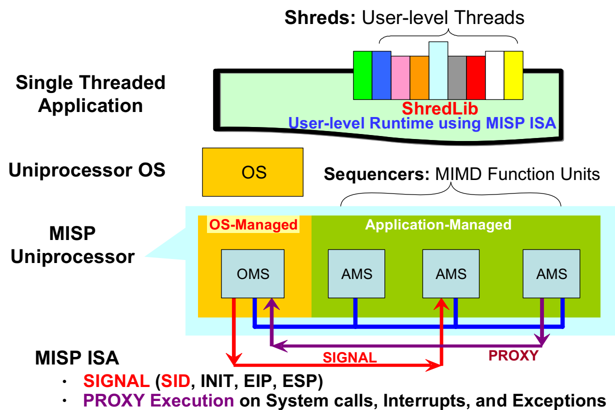

Figure 1. MISP Uniprocessor (4 Sequencers) and Execution Environment

sequencers to physical sequencers is called *sequencer virtualization*. For the rest of this paper, and without loss of generality, we focus on sequencers that are of the same ISA family but with potential differences in microarchitecture organization, performance, power and degree of functional completeness. We also assume a trivial one-to-one mapping between virtual sequencers and physical sequencers, and leave a full analysis of sequencer virtualization to future work.

2.2 OS-managed Sequencer and Application-managed Sequencer

Contemporary OSs require processor support for multiple privilege levels of instruction execution, so that the OS kernel services and application software have different degrees of access privilege and protection for various processor resources. In IA-32, the privilege levels are also referred to as Rings, and most OS kernel services execute at Ring 0 while the application (user-level) code executes at Ring 3.

In an IA-32 MISP processor, the OS-managed sequencer (OMS) supports the full IA-32 ISA semantics at all privilege levels, and thus is capable of executing both OS kernel services and user-level code. In contrast, the application-managed sequencers (AMS) are user-level resources that execute only in Ring 3 and, therefore, need to support only the Ring 3 subset of the IA-32 ISA semantics. Without loss of generality, we assume that AMSs support the Ring 3 ISA semantics for the discussion that follows.

From the OS's perspective, one thread can be scheduled to one MISP processor at a time, and the AMSs effectively serve as MIMD "functional units" to the OMS. Upon a thread context switch, the cumulative context states for all sequencers are subject to being saved and restored in the OS kernel. In fact, providing the aggregate save area for the cumulative states of all AMSs is the primary, if not the only, additional OS support required of a legacy OS.

Allowing asymmetry between the OMS and its AMSs promises opportunities for much simpler and more efficient microarchitectural implementations of AMSs by shedding a portion of the ISA legacy. However, care should be taken to avoid impairing the well established shared-memory multithreaded programming paradigm, and to ensure the appearance of functional symmetry amongst the sequencers. To this end, MISP provides direct architectural support to preserve the shared-memory programming paradigm and gives software the illusion of functional symmetry amongst sequencers.

2.3 Architectural Support for Sharing the Virtual Address Space

In IA-32, a small set of Ring 0 control registers are used to manage the virtual to physical memory address translation (e.g. the page table base is stored in control register CR3). To support a shared virtual address space among sequencers in a MISP processor, all sequencers are required to synchronize upon any update to this set of Ring 0 control registers. A variety of approaches can be used to efficiently support this mechanism. For example, the simplest implementation suspends all AMSs whenever the OMS performs a transition from Ring 3 to Ring 0, and ensures any OMS's update to the related control registers be synchronized across all AMSs before the subsequent OMS's transition from Ring 0 back to Ring 3. Once the OMS has returned back to Ring 3, the AMSs can resume execution. In modern IA-32 implementations, any write to control register CR3 automatically purges that sequencer's TLB. As long as there is no page fault (page fault handling is described in Section 2.5), each sequencer can independently execute a shred in Ring 3 in parallel with other sequencers, with any TLB miss handled independently by the sequencer's hardware TLB page walker.

Pausing the AMSs while the OMS executes in Ring 0 ensures correctness of execution, but this delay is unnecessary when the OMS does not modify the control registers that manage the application's virtual address space. Instead of pausing the AMSs, a more aggressive microarchitecture could allow the AMSs to continue executing speculatively while monitoring for any control-register updates from the OMS. If no related control registers are updated while the OMS executes in Ring 0, the AMSs can retire the speculatively executed instructions once the OMS returns to Ring 3. However, such an approach could require much higher hardware complexity. To further assess the performance impact of synchrony induced by the OMS ring transition, we will present findings from a prototype-based evaluation in Section 5.

IEEE COMPUTER SOCIETY

2.4 Architectural Support for Inter-Sequencer Signaling

In the classic SMP system, processors communicate via the interprocessor interrupt (IPI) mechanism. To support concurrent execution of multiple instruction streams on all the sequencers in a MISP processor, the MISP ISA introduces a user-level dual of the interprocessor interrupt (IPI).

MISP defines a SIGNAL instruction, which at minimum includes a Sequencer ID (SID), a starting instruction pointer (EIP), and a stack pointer (ESP) as operands. SIDs are logical identifiers that specify a particular sequencer within the current MISP processor, and the ⟨EIP, ESP⟩ pair forms a shred continuation. SIGNAL generates an inter-sequencer signal that delivers the shred continuation ⟨EIP, ESP⟩ to the destination sequencer denoted by the SID. To initiate a shred on an AMS, the application executes the `SIGNAL(sid, eip, esp)` instruction to send an egress signal to the AMS. The OMS then continues executing at the instruction following the SIGNAL instruction. Once the AMS receives the signal and commences the shred at the continuation, the AMS and the OMS are executing two shreds in parallel.

With the SIGNAL instruction, the MISP ISA has a general mechanism to perform egress control and data transfer between sequencers in a MISP processor. To respond to an ingress inter-sequencer signal, the destination sequencer supports asynchronous control transfer via a YIELD-CONDITIONAL mechanism, which is similar to the one described in the Virtual-Multithreading (VMT) work [26]. With YIELD-CONDITIONAL, a sequencer can set up a trigger-response mapping between an ingress inter-sequencer signal and a corresponding handler. When the anticipated asynchronous event occurs, the shred effectively performs an asynchronous function call to the handler: the sequencer suspends execution of the current shred, saves the next EIP, and then performs a fly-weight control transfer into the handler. Later, upon return from the handler, the original shred is resumed from the saved EIP.

2.5 Architectural Support for Proxy Execution

MISP introduces a novel architectural mechanism, called *proxy execution*, which provides software the illusion of functional symmetry amongst the sequencers, despite any physical asymmetry or heterogeneity that may exist. The main idea of proxy execution is to architecturally define a set of triggering conditions that cause a user-level fault-type exception to occur on an AMS. When one of the triggering conditions is encountered, the fault exception is relayed from the AMS to the OMS. Upon notification, the OMS can perform an asynchronous control transfer to suspend its current code execution and jump to a proxy handler to perform proxy operations on behalf of the AMS. During proxy execution, the OMS saves its original sequencer states and temporarily assumes the sequencer states of the faulting AMS. The OMS re-executes the faulting instruction, triggering the fault again and causing OS services to be activated — the very work that cannot be done on the AMS. After the fault is handled by the OS, the proxy handler saves the current OMS states and then restores them to the AMS's context, effectively resuming the original AMS shred execution on the AMS. The handler then restores the original OMS states back to the OMS context, effectively resuming execution of the interrupted OMS. The OMS and AMS are once again running in parallel.

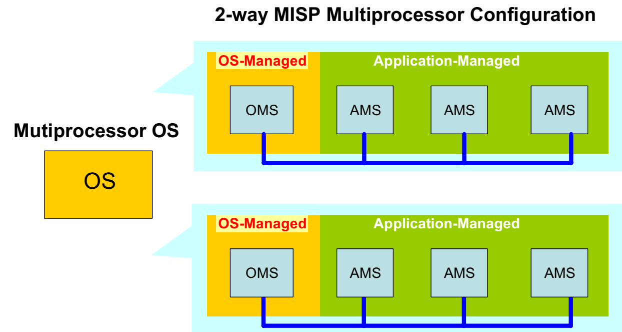

Figure 2. MISP Multiprocessor

The basic triggering conditions that can lead to proxy execution are exceptions (e.g. page faults) and OS service requests (e.g. system calls) encountered on an AMS. These events require the OS's attention, but the AMS cannot directly attain the OS services, so the proxy execution mechanism enlists the OMS as a temporary helper sequencer to perform the OS services on behalf of the AMS. From another perspective, the proxy mechanism provides architectural support for dynamic shred migration, and therefore, to the application, the AMS appears to be functionally equivalent to the OMS. The proxy execution mechanism is essentially an architectural guarantee for any shred to make forward progress regardless of which sequencer it is running on. It is important to note that the above algorithmic steps for proxy execution can be used to deal with any AMS proxy request. In other words, at minimum, a single proxy handler on the OMS is sufficient to deal with all proxy conditions.

2.6 MISP MP Architecture

Like traditional processors, multiple MISP processors can be combined to form a multiprocessor system. As illustrated in Figure 2, the OS sees only the OMSs and schedules threads to run on each. Each thread, in turn, can use the

0-7695-2608-X/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

SIGNAL instruction to utilize an AMS within the thread's MISP processor. If a thread is context-switched away by the OS, its AMSs are suspended and their states are saved to memory. If that thread is later re-scheduled to run on a different OMS, the previously saved AMS states are restored on the new AMSs. If one OMS transitions into Ring 0, it suspends its AMSs as before, but the second MISP processor is unaffected and continues running.

To the OS, the MISP MP system appears as a normal SMP system, and is managed as such. Conveniently, the semantics of instructions that may affect multiple processors, such as IA-32's WBINVD instruction [13], remain valid for both OMSs and AMSs. As a result, legacy SMP OS mechanisms, such as the TLB shootdown protocol [14], are supported without change to the OS. Recall that a MISP processor synchronizes its AMSs with the OMS to maintain a shared virtual address space. Similarly, the OS maintains shared virtual address spaces across active OS threads by synchronizing the threads upon any changes. To illustrate this concept, an application can have two threads, each running on a separate MISP processor, as shown in Figure 2. If the OS changes the virtual address space for the running process, the OS must notify all active threads in that process. The OS sends an IPI to each currently active thread, causing each to trap into the OS (a Ring 3 to Ring 0 transition). The OS can then safely manage the virtual address space for the entire application process. Since the AMSs will necessarily be updated when the OMS returns from Ring 3, the virtual address space remains consistent across all sequencers in the MISP processor.

3. MISP Programming Model

Like an OS thread, a shred represents a stream of instructions that can execute concurrently with other instruction streams. However, since shreds belong to an OS thread, when the OS thread is context switched, the shreds are effectively suspended. So a shred bears more resemblance to the fiber abstraction provided by Microsoft Windows [21]. Fibers and shreds are each created and managed by the application. Shreds, however, differ from fibers in a significant way. Within an OS thread, only one fiber can be scheduled for execution at any time, which prevents a thread's fibers from executing concurrently. In contrast, an OS thread's shreds can execute concurrently on multiple sequencers. In other words, shreds are like concurrently executing fibers.

Because an application can not rely on the OS for shred scheduling, such services must be provided at application level. Since the MISP architecture supports the shared-memory multithreaded programming paradigm, a POSIX-compliant user-level runtime for shred control and synchronization can be built to provide commonly used shred scheduling mechanisms. The runtime provides seamless integration between the MISP programming model and the threading model commonly provided by the OS.

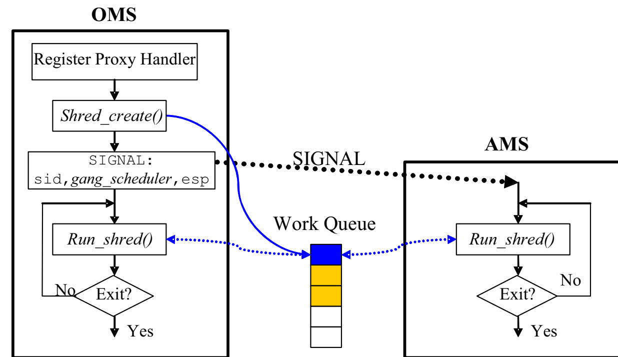

Figure 3. MISP-based Shred Gang Scheduling via Shared Work Queue

To illustrate the MISP programming model, the following example demonstrates how to create an $M : N$ (shreds to sequencers, $M >= N$) gang scheduler based on a simple work queue. The example in Figure 3 shows a simple algorithm for scheduling shreds in FIFO order to execute concurrently on the OMS and one AMS. The work queue is a mutex-protected shared memory data structure, and holds the shred continuations (i.e. $\langle$EIP, ESP$\rangle$ pairs) that are ready to execute. The algorithm works as follows: The OS thread executing on the OMS first uses the trigger-response mechanism to register a proxy-handler routine for servicing proxy requests from the AMSs. Next, the thread creates a new shred by calling the *Shred_create* routine, which inserts a new shred continuation into the work queue. The thread then initiates a gang scheduler shred on the AMS through the SIGNAL instruction, and starts the gang scheduler routine on the OMS through a function call. At this point, gang scheduler routines are running concurrently on the two sequencers and contending for access to the work queue. Inside each gang scheduler, the *Run_shred* routine interrogates the mutex to the work queue, attempts to grab an available shred and, if available, performs a light-weight context switch to execute the shred. One gang scheduler will eventually obtain the mutex to the work queue, retrieve the first shred and perform the context switch to run the shred. Once the shred is executing, it can create additional shreds via *Shred_create*. Once a shred completes or yields, execution returns to the gang scheduler, which will again contend for the next shred from the work queue. This process continues until all shreds have completed execution.

A shred running on a sequencer can choose to voluntarily yield and place itself back into the work queue, causing a different shred to be chosen from the queue and scheduled on the sequencer. By supporting yield semantics, a variety

IEEE COMPUTER SOCIETY

of shred synchronization primitives can be introduced, including various queue-based locking algorithms.

4. MISP Prototype System

We have built a prototype MISP processor using special firmware and developed a user-level multi-shredding runtime, called *ShredLib*. Together, the prototype and runtime provide an effective software development vehicle facilitating the development of multi-shredded applications, the migration of existing multithreaded applications from an SMP platform target to the MISP processor, the analysis of MISP performance, and the assessment of key architectural tradeoffs of the MISP architecture.

4.1 Emulator of the MISP Architecture

Our MISP processor prototype is built on a 4-socket SMP platform with 3.0 GHz Intel IA-32 dual-core processors. HyperThreading is disabled to simplify the performance analysis. The operating system is Windows Server 2003. By setting the `/NUMPROC` parameter to 1 in the `boot.ini` configuration file, Windows recognizes and boots on only one processor, giving our system the appearance of a MISP uniprocessor, with one processor OS-visible, and the other seven sequestered. We use the OS-visible processor to emulate an OMS and the sequestered processors to emulate seven AMSs.

With a custom firmware, all key aspects of the MISP ISA extensions are emulated, including the SIGNAL instruction, the sequencer-specific state machines for inter-sequencer signaling, the YIELD-CONDITIONAL mechanism to register a proxy handler (which catches an AMS exception condition via the trigger-response mechanism) and to handle the asynchronous control transfer, and the detection of the proxy execution triggering conditions on AMSs. The firmware also detects ring level transitions on all sequencers. The firmware suspends all AMSs when the OMS transitions from Ring 3 to Ring 0, and resumes the AMSs once the OMS returns from Ring 0 to Ring 3. In the case of proxy execution, the firmware provides a Ring 3 inter-sequencer signal requesting that the AMS save its state context to memory, which provides the OMS the state needed to impersonate the AMS.

The firmware also facilitates detailed performance analysis by providing coarse- and fine-grain event logging. The coarse-grained event logging provides total counts for the number and cause of ring transitions on each sequencer. The fine-grained event log provides time-stamped events indicating the start and end time of a particular event; as an example, if a page fault occurs on the OMS, the fine-grained event log provides the time spent in the firmware managing the AMSs and the time spent in the OS servicing the page fault. The firmware also provides feedback to the application developer on the number of proxy execution events and their causes. Developers can use this information to tune their applications by identifying where significant time is spent handling proxy execution.

4.2 Prototype Runtime Environment

The ShredLib runtime is a dynamically linked library (DLL) that implements the shared-memory multi-shredded programming model for the MISP architecture. By default, ShredLib implements a POSIX-compliant suite of shred control and shred synchronization primitives, including support for critical sections, mutexes, condition variables, semaphores, and events. ShredLib also provides a generic routine to handle proxy execution for all proxy triggering conditions. To facilitate migration of legacy multithreaded applications to a MISP processor, ShredLib provides legacy API translations for the Pthreads [6] and Win32 Threads [4] APIs. ShredLib also seamlessly supports both Thread Local Storage [21] and Structured Exception Handling [21] — two common Windows OS services supported by Intel and Microsoft compilers (e.g. `__declspec(thread)`) — for shreds, without requiring recompilation or changes to the compiler.

Internally, ShredLib implements several different shred scheduling algorithms and can be customized for an application's special requirements. ShredLib also provides a detailed event logging system that can profile relevant scheduling activities, such as inter-shred dependencies and contention on common synchronization objects. This event logging system is complementary to that provided by the prototype MISP processor's custom firmware.

5. Experimental Results

To study the MISP architecture, we selected existing multithreaded applications, converted the applications to use shreds, and then executed these applications on several different configurations of the MISP prototype (described in Section 4). In the analysis that follows, we first focus on quantifying the performance overhead of synchronizing the AMSs during ring transitions, as this operation can potentially degrade performance. We then investigate how legacy single- and multi-threaded applications will co-exist with multi-shredded applications. Finally, we measure empirically how easily MISP can be used by application programs. Our initial analysis of the MISP architecture shows that:

- Application performance is relatively insensitive to the overhead of synchronizing AMSs during privilege level transitions (Section 5.3).

IEEE COMPUTER SOCIETY

- A MISP-based multiprocessor system can improve throughput over a single MISP when used in a multiprogramming environment (Section 5.4).
- Porting multithreaded application software from SMP to the MISP architecture requires little effort and few, if any, structural changes to the original source code (Section 5.5).

Throughout this evaluation, we compare and contrast MISP performance with that of an equivalently configured SMP system. MISP is not designed to replace SMP and the comparison with SMP should not be confused as such; in fact, as we show in Section 5.4, MISP and SMP are complementary architectures. SMP performance simply provides a convenient baseline from which the performance impact of MISP architectural tradeoffs can be better understood.

5.1 MISP Overhead Modeling

MISP introduces an overhead of synchrony not found in SMP architectures. All AMSs must suspend execution when the OMS transitions from Ring 3 to Ring 0, and before resuming, each AMS must synchronize with the OMS on a small set of Ring 0 control registers once the OMS transitions back to Ring 3. This synchrony overhead can be separated into three categories: the serialization time across OMS ring transitions, the overhead associated with each shred to perform proxy execution, and the overhead for the OMS to handle proxy execution. Each category of performance overhead is modeled below. In each of the models, $signal$ represents the time needed for inter-sequencer communication.

Serialization Overhead Across Ring Transitions

$$serialize = 2 \times signal + priv \tag{1}$$

Recall that the prototype suspends all AMSs when the OMS transitions from Ring 3 to Ring 0, as it implements a simple mechanism to synchronize the privileged state. Equation 1 indicates the performance overhead as the cost to simultaneously SIGNAL all AMSs to suspend themselves ($signal$), plus the time spent executing in the OS ($priv$) and the cost to SIGNAL all AMSs to resume themselves ($signal$). Note that $priv$ includes the time needed for the AMSs to save and restore their state upon an OS thread context switch; we expect this operation to be performed concurrently for all AMSs.

AMS Proxy Execution Overhead

$$proxy_{egress} = 3 \times signal \tag{2}$$

Equation 2 indicates the performance overhead incurred by a shred that requires proxy execution. This overhead includes the cost to notify the OMS of the proxy request ($signal$), the time for the OMS to suspend all active AMSs ($signal$), and the time for the OMS to resume all AMSs after proxy execution ($signal$).

OMS Proxy Execution Overhead

$$proxy_{ingress} = signal + serialize \tag{3}$$

Equation 3 indicates the overhead incurred by the OMS to handle the proxy execution request. This overhead equals the time to process the proxy request from the AMS ($signal$) plus the cost of a serialization operation ($serialize$, from Equation 1).

5.2 Workloads and Methodology

To analyze the performance of the MISP architecture, we choose a number of compute-bound, multithreaded kernels and applications from the SPEComp benchmark suite [1] and the Recognition-Mining-Synthesis (RMS) benchmark suite [7]. Each application is converted to use shreds in place of threads and then executed on the MISP prototype system. The SPEComp benchmark suite represents real multithreaded scientific applications; from this suite we select the applications *swim, applu, galgel, equake* and *art*, and use the reference input set (ref) as input. The RMS suite includes kernels of code extracted from emerging applications; for example, the suite includes kernels of code for matrix multiplication (both dense and sparse), partial differential equations solver (Gauss-Seidel iterative solver), and K-Means clustering. It also includes larger applications, such as a highly scalable multithreaded graphics application, RayTracer [10]. All applications are compute-bound and exhibit varying degrees of scalability, and each is compiled with the Intel® C++ or Intel® Fortran compilers. The RMS benchmark applications require a recompilation from source code to use the ShredLib API for shred management. The SPEComp applications use a MISP-enabled version of OpenMP runtime from the Intel® Threading Tools [11] and, thus, none require source code modifications.

While applications execute much faster on the MISP processor prototype than in a software-based simulator, the prototype still incurs an emulation overhead far greater than the performance overhead expected on an actual MISP processor. To account for emulation overhead, we measure the time spent executing firmware and subtract this time from the total execution time, as this firmware would not be present in a real implementation. For the cost of $signal$, we assume 5000 clock cycles, representing what we believe to be a conservative estimate of a microcode-based implementation of the inter-sequencer signaling mechanism.

IEEE COMPUTER SOCIETY

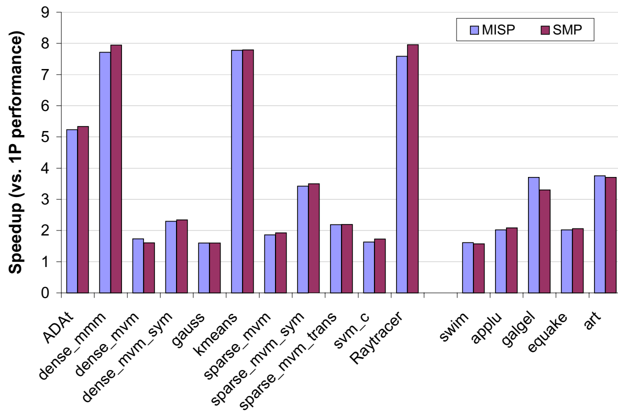

Figure 4. MISP Performance: 1 OMS + 7 AMS

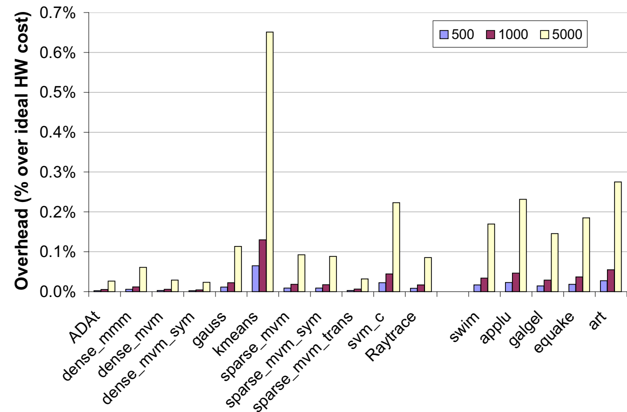

Figure 5. Sensitivity to Signal Cost

5.3 MISP Performance Analysis

We first examine the performance tradeoffs of a MISP uniprocessor. The prototype system contains eight sequencers: one sequencer is configured as the OMS and the remaining seven as the AMSs. Our primary focus is to assess the performance impact of synchrony due to the ring transitions of the OMS. The overhead of each serializing event potentially increases as the number of AMSs increases, as parallelism is lost while servicing the event.

Figure 4 shows, for each application, MISP performance as speedup over single sequencer performance. For comparison, we also show the performance for those same applications when executing on a similarly configured SMP machine with eight cores. The RMS applications perform, on average, 1.5% slower on MISP than their performance on the SMP system, while the SPEComp applications perform, on average, 1.9% faster on MISP. The performance data indicate that suspending all AMSs during privileged code execution has little practical effect on the performance of these benchmarks.

We further analyze the origins of these serialization events to help understand the impact of synchrony on performance. Table 1 summarizes statistics for all salient architectural events that cause the MISP processor to serialize execution to synchronize privileged state across all AMSs. The table shows the average number of serializing events (events that cause privilege level transitions) that occur during program execution. The events are separated into those occurring on the OMS and those occurring on the AMSs.

From Table 1, we see that there is an opportunity to optimize away many of the proxy execution events, further reducing the serialization time. For all of the RMS applications and all but one of the SPEComp applications (*galgel*), compulsory page faults cause the majority of proxy execution events. Once the working set of memory pages are resident in physical memory, the AMSs make no further proxy execution requests. If the OMS probes each page (e.g. by accessing a single byte within the page) while executing in the serial region of code that precedes parallel execution, the number of proxy execution events for page faults can be significantly reduced. Future MISP runtime software or compilers potentially can perform such optimization.

A conservative estimate for the overhead in a microcode-based implementation of inter-sequencer communication is 5000 cycles. We now assess how much additional performance can be realized if a more aggressive hardware optimization is used to reduce the inter-sequencer signaling cost. To this end, we assume an ideal hardware implementation with zero-cost for $signal$ as baseline and consider two other configurations with overheads of 500-cycle and 1000-cycle, respectively. We calculate the additional OMS overhead by first separating the events into those that originate on the OMS and those that originate on an AMS. For those events that originate on the OMS, Equation 1 indicates the overhead. For those events that originate on an AMS, Equation 2 indicates the total overhead. Figure 5 shows the performance delta normalized to the baseline. The 5000-cycle implementation performs only 0.15% slower, on average, and 0.65% slower in the worst case (Kmeans), than the ideal hardware implementation. Clearly, the throughput performance of the applications is insensitive to the overhead of the inter-sequencer signaling.

5.4 MISP Multiprocessor Performance

The MISP multiprocessor system is designed to improve throughput performance for workloads that include single- and multi-threaded applications. As an initial exploration of the MISP MP design space, we evaluate performance while varying the ratio of AMSs to OMSs in the system, but maintaining the total number of sequencers at eight. Figure 6 shows the multiprocessor MISP configurations that we examined. In the figure, the 4×2 configuration consists of four

0-7695-2608-X/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

Table 1. Serializing Events

Application	OMS Events				Total AMS Events	
	SysCall	PF	Timer	Interrupt	SysCall	PF
RMS						
ADAt	0	1	168	20	0	9
dense_mmm	0	29	141	15	0	133
dense_mvm	0	1	64	5	0	5
dense_mvm_sym	0	2	1178	104	0	9
gauss	8	7170	1736	158	0	1
kmeans	8	7170	260	25	0	2
sparse_mvm	0	27	114	13	0	205
sparse_mvm_sym	0	11	343	31	0	669
sparse_mvm_trans	0	26	826	75	0	200
svm_c	8	7204	1006	101	0	1307
RayTracer	0	210	591	66	0	979
SPEComp						
swim	77,009	59,570	96,687	10,281	0	346,201
applu	1,394	59,540	57,282	5,115	0	327,313
galgel	881	152,806	64,880	6,242	0	140,180
equake	45,937	47,896	29,727	3,093	0	85,654
art	19,978	133,672	31,647	2,923	436	138,464

SysCall: the number of traps to the OS **PF:** the number of page faults incurred
Timer: the number of timer-clock interrupts **Interrupt:** the remaining uncategorized interrupts

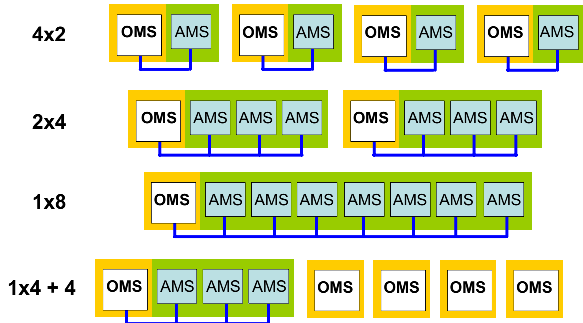

Figure 6. MISP MP Configurations

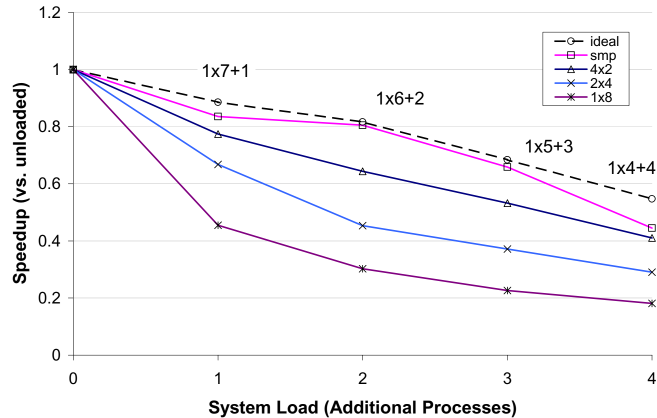

Figure 7. MISP MP Performance

MISP processors, where each MISP processor contains one OMS and one AMS. Likewise, the 2×4 configuration consists of two MISP processors, where each MISP processor contains one OMS and three AMSs. The $1 \times 4 + 4$ configuration illustrates an uneven partitioning of AMSs to OMSs. It consists of one MISP processor that contains one OMS and three AMSs, plus four single sequencer processors (1 OMS and no AMS).

Figure 7 shows the performance of RayTracer as non-shredded applications are gradually added to the system. For the 1×8 configuration, the performance of RayTracer decreases nearly linearly as the number of competing processes increases. This is because the single-threaded application must share the OMS with the shredded application and, as a result, the AMSs are utilized only 50% of the time. As we increase the number of MISP processors and decrease the number of AMSs assigned to each OMS, scaling performance improves. This is because more sequencers are being utilized at a given time. Still, AMSs will remain underutilized as long as the thread running on the OMS does not create shreds.

The optimal configuration depends on the characteristics of the workload. For the most efficient use of available resources, the non-shredded applications should execute on OMSs that have no AMSs. This optimal configuration is shown as the "ideal" trend. Note that the ideal configurations are uneven in the number of AMSs associated with each OMS. This imbalance may complicate shred scheduling, since a thread (and its shreds) should not migrate to a MISP processor that does not have the proper number

0-7695-2608-X/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

Table 2. Applications Ported to the MISP Architecture

Application	Description	Effort (days)
Intel® Thread Checker [11]	Identifies errors in multithreaded applications	5
Intel® Thread Profiler [11]	Provides performance analysis for multithreaded applications	5
Intel® OpenMP Library [24]	Intel's implementation of the OpenMP specification, included with Intel® compilers	5
RayTracer [10]	Research prototype for studying Ray Tracing algorithms	1
Open Dynamics Engine [22]	Physics modeling engine, multithreaded in-house	3
Media Encoder	Commercial multithreaded MPEG video encoder	13
Lame-MT [20]	Multithreaded MPEG-1 Layer 3 (MP3) encoder	0.5
BEA JRockit® [3]	High-performance, commercial Java Virtual Machine	15
RMS Benchmark Suite [7]	Multithreaded kernels from emerging Recognition-Mining-Synthesis workloads	0.5 each

of AMSs. Also, a dynamic workload consisting of many single-threaded, multi-threaded and multi-shredded applications may not perform well on a static uneven configuration. This effect motivates future investigation into techniques for dynamically binding AMSs to OMSs, even to the extent of crossing socket boundaries, within a multiprocessor MISP system. Such techniques are enabled through sequencer virtualization, a subject beyond the scope of this paper.

5.5 MISP Programmability Assessment

During the process of developing and validating the MISP prototype, we converted a number of multithreaded applications to be multi-shredded. The effort expended provides a good indicator of the difficulty in programming for this architecture. Each application is ported by recompiling it to use ShredLib's API support for Win32 Threads or Pthreads. Table 2 summarizes the estimated porting time for each application.

Most applications required very few code changes to replace threads with shreds, since ShredLib provides a thread-to-shred API mapping. With most applications, we simply changed the application's source code to include a single header file that contains ShredLib's thread-to-shred API mapping, and then recompiled. Only one application, the Open Dynamics Engine, required a structural modification to the code. Simply converting all threads to shreds resulted in an inefficient use of the AMSs, as the main program thread sleeps inside of the OS while waiting on the user to provide input. By using a native OS thread to handle user I/O and a separate native OS thread consisting of multiple shreds to perform the compute-intensive parallelized computation, the AMSs were more efficiently utilized.

6. Related Work

Historically, all ISAs have assumed the sequential execution model, or SISD (single-instruction single-data) execution. Over time, ISAs have incrementally evolved to expose new computation resources to application software so that programmers can directly manage the architectural resources to enhance performance. In recent years, SIMD (single-instruction multiple-data) extensions have been added to most ISAs to exploit fine-grained data parallelism present in many multi-media applications, but these SIMD extensions are still restricted to a single instruction stream and cannot directly exploit thread level parallelism. Examples include the three Streaming SIMD Extensions to IA-32: SSE, SSE2 and SSE3 [12]. This paper proposes a MIMD (multiple-instruction multiple-data) extension to the IA-32 ISA to take advantage of the abundant execution capability of multi-core systems. This new extension is a natural progression of the ISA extensions, as depicted in Figure 8, for exploiting parallelism.

In the classic SMP system, the interprocessor interrupt (IPI) mechanism is the fundamental architectural support needed by the multiprocessor OS kernel to manage all processors and threads. IPI is a privileged architectural mechanism for interprocessor signaling and asynchronous control transfer, so it can only be used by the privileged software layer (e.g. the OS kernel and device drivers), and is inaccessible to user-level application. The MISP ISA introduces application-managed sequencers as architecturally distinct resources and provides a user-level dual to the IPI for inter-sequencer signaling and asynchronous control transfer between sequencer resources. With this user-level IPI-like mechanism, true user-level threads can be created.

While few details are publicly known, the MIPS multithreading application-specific extension (MT ASE) [16], appears to combine classic multithreading concepts with ISA semantics. In particular, the MT ASE defines a virtual processing element (VPE) to appear as a processor, so that a system with multiple VPEs would appear to the OS as multiple CPUs. MT ASE provides explicit instructions for thread control, including thread creation, yield and termination. MISP differs from the MT ASE in several salient ways. First, while the VPE is an abstraction of a processor and resembles the sequencer abstraction in MISP, the VPE is made visible to the OS as a fully

IEEE COMPUTER SOCIETY

OS-managed resource. In MISP, a sequencer can be OS-managed or application-managed. Second, the MT ASE requires dedicated hardware resources to maintain thread context and manage thread scheduling. In contrast, MISP defines only an inter-sequencer signaling and asynchronous control transfer mechanism. Our canonical approach allows for a much simpler hardware implementation, and leaves thread control and synchronization to application software.

There have been many research proposals for ISA extensions to support various forms of hardware multithreading. MISC [25] described a message-passing based hardware mechanism where parallel execution of multiple instruction streams can be orchestrated. Compared to MISP, MISC seems to require significantly more hardware complexity in providing sophisticated support to point-to-point interconnect between processing elements and between processing elements and the memory subsystem. In addition, MISC provides various special hardware supports to sentinel and vector instructions to harvest more data parallelism from single application. Another active area of extensive research is speculative multithreading, which is a special form of MIMD parallelism used to accelerate single thread application performance [19, 23, 27]. MISP is largely orthogonal to these proposals: While MISP's inter-sequencer signaling and asynchronous control transfer mechanism can be used by speculative threads, MISP itself does not provide any additional architectural support to manage speculative states.

Recently, the CELL architecture has garnered an extensive amount of attention, in particular, with respect to its unique heterogeneous MIMD architecture and its divergence from the shared-memory programming model [8]. CELL directly exposes the physical sequencer hardware of different ISAs (one PowerPC PPE and multiple streaming processors, called SPEs) to the programmer, and its inter-sequencer signaling mechanism resembles an interprocessor interrupt (IPI). Instead of providing direct architectural support for the shared-memory programming model, each SPE in CELL has its own local memory and interacts with the other processors via a globally coherent DMA engine. As discussed in [8], CELL's programming model imposes a significant burden on the programmer since it requires code and data to be partitioned into PPE and SPE portions, and it also requires explicit transfer of code and data between system memory and the local store. To give the appearance of a shared-memory multithreaded programming model, sophisticated compiler and runtime support are needed to emulate cache coherence in software and orchestrate DMA data transfers. In contrast to CELL, the MISP architecture supports the shared-memory programming model across all sequencers, which greatly simplifies application development. Also, MISP provides architectural support for proxy execution, which allows one sequencer to execute code on behalf of another. Through the proxy execution mechanism,

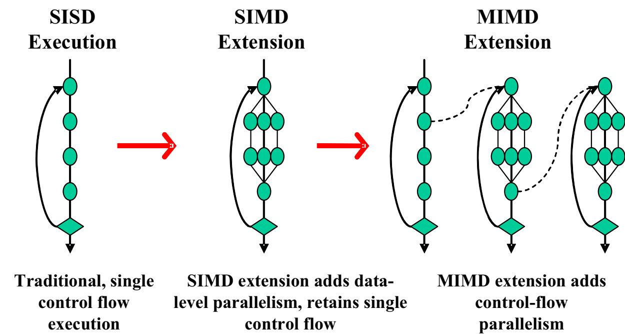

Figure 8. Progression to MIMD ISA Extension

MISP ensures that asymmetric sequencers appear as functionally symmetric resources to the application. In contrast, the CELL SPE does not support such a mechanism; therefore, the asymmetry is exposed to the programmer and the SPEs can not execute general purpose code. For general purpose ISAs, like the IA-32 family, proxy execution is absolutely necessary to support legacy software.

7. Conclusions and Future Work

This paper presents a novel ISA-based approach by which multithreaded application programs directly manage multi-core processor resources, including asymmetric or heterogeneous cores, using the well established multithreaded programming model. Through an extensive prototyping effort using a physical IA-32-based SMP system with custom firmware, we quantify some key architecture tradeoffs for MISP and demonstrate the feasibility to realize a simple and efficient implementation of the MISP architectural extension to the IA-32 family of processors. The minimal effort required to port a production quality threading toolset and a set of key multithreaded workloads to the MISP prototype indicates that maintaining the multithreaded programming model greatly eases software development for the MISP platform.

Clearly there are many interesting directions for future research. This paper is the first step in developing the concept of MISP and demonstrating its feasibility. For future work, we will investigate opportunities to exploit functional and performance asymmetry for efficient MISP design, and more judicious shred scheduling algorithms in the ShredLib user-level runtime. In addition, as indicated in Section 5.4, it is potentially beneficial to harvest both compute-bound TLP via multi-shredding and I/O-bound TLP via multithreading. To this end, we plan to investigate architecture support to MISP multiprocessors such as sequencer virtualization techniques that can judiciously map physical sequencers to multiple logical OMSs or AMSs.

IEEE COMPUTER SOCIETY

8. Acknowledgements

We acknowledge productive collaboration with Dion Rodgers, Baiju Patel, Prashant Sethi, Sanjiv Shah, Paul Petersen, Dave Poulsen, Grant Haab, Shirish Aundhe, Suresh Srinivas, John Reid and Xinmin Tian. In addition, we thank the anonymous reviewers whose inputs have helped the authors greatly improve the quality of this paper.

References

[1] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Proceedings of the International Workshop on OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming*, pages 1–10, 2001.

[2] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 506–517, Washington, DC, USA, 2005. IEEE Computer Society.

[3] BEA Systems. BEA JRockit®: High performance on Intel®platforms. `http://www.bea.com/framework.jsp?CNT=index.htm\&FP=/content/products/jrockit`.

[4] J. Beveridge and B. Wiener. *Multithreading Applications in Win32: The Complete Guide to Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[5] R. Bryant, Barnes, J. Hawkes, J. Higdon, and J. Steiner. Scaling Linux to the Extreme: From 64 to 512 Processors. In *Ottawa Linux Symposium*, pages 133–148, 2004.

[6] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[7] P. Dubey. Recognition, Mining and Synthesis Moves Computers to the Era of Tera. *Technology@Intel Magazine. `http://www.intel.com/technology/magazine/computing/recognition-mining-synthesis-0205.pdf`*, pages 1–10, February 2005.

[8] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing Compiler for the CELL Processor. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[9] E. Grochowski, R. Ronen, J. Shen, and H. Wang. Best of Both Latency and Throughput. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 236–243, Washington, DC, USA, 2004. IEEE Computer Society.

[10] J. Hurley. Ray Tracing Goes Mainstream. *Intel Technology Journal*, May 2005.

[11] Intel Corporation. Intel®Threading Tools. `http://www.intel.com/cd/software/products/asmo-na/eng/threading/index.htm`.

[12] Intel Corporation. *IA-32 Intel® Architecture Software Developer's Manual Volume 1: Basic Architecture*, June 2005.

[13] Intel Corporation. *IA-32 Intel® Architecture Software Developer's Manual Volume 2B: Instruction Set Reference, N–Z*, January 2006.

[14] Intel Corporation. *IA-32 Intel® Architecture Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, January 2006.

[15] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, Washington, DC, USA, 2003. IEEE Computer Society.

[16] M. Levy. Multithreaded Technologies Disclosed at MPF: Imagination and MIPS Discuss Details of New Architectures. In *Microprocessor Report*, 2003.

[17] J. Li and J. F. Martinez. Power-Performance Implications of Thread-level Parallelism in Chip Multiprocessors. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2005.

[18] T. Morad, U. Weiser, and A. Kolodny. ACCMP — Asymmetric Chip Multi-Processing. Technical Report 488, CCIT Technical Report, June 2004.

[19] E. Özer, T. M. Conte, and S. Sharma. Weld: A Multithreading Technique Towards Latency-Tolerant VLIW Processors. In *HiPC*, pages 192–203, 2001.

[20] G. Raichshtain. LAME-MT. `http://softlab.technion.ac.il/project/LAME/html/lame.html`.

[21] J. Richter. *Programming Applications for Microsoft Windows, Fourth Edition*. Microsoft Press, Redmond, WA, USA, 1999.

[22] R. Smith. Open Dynamics Engine. `http://www.ode.org`.

[23] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, New York, NY, USA, 1995. ACM Press.

[24] X. Tian, M. Girkar, S. Shah, D. Armstrong, E. Su, and P. Petersen. Compiler and Runtime Support for Running OpenMP Programs on Pentium®- and Itanium®-Architectures. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003.

[25] G. Tyson, M. Farrens, and A. R. Pleszkun. MISC: a Multiple Instruction Stream Computer. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 193–196, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[26] P. H. Wang, J. D. Collins, H. Wang, D. Kim, B. Greene, K.-M. Chan, A. B. Yunus, T. Sych, S. F. Moore, and J. P. Shen. Helper Threads via Virtual Multithreading on an Experimental Itanium® 2 Processor-based Platform. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 144–155, New York, NY, USA, 2004. ACM Press.

[27] A. Wolfe and J. P. Shen. A Variable Instruction Stream Extension to the VLIW Architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–14, New York, NY, USA, 1991. ACM Press.

0-7695-2608-X/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY