Computer Sciences Department

Metadata-Based Parallelization of Program Instrumentation

Matthew D. Allen
Gurindar S. Sohi

Technical Report #1608

August 2007 (Revised March 2008)

Metadata-Based Parallelization of Program Instrumentation

Matthew D. Allen
matthew@cs.wisc.edu

Gurindar S. Sohi
sohi@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison

ABSTRACT

Program instrumentation has a wide variety of useful applications, but tool writers must overcome the challenge of substantial overheads caused by introducing additional code and data into a program. This paper observes that instrumentation usually operates on many discrete, independent data structures, which we call *metadata parallelism*. We proposes to exploit this phenomenon to reduce the overhead of instrumented programs by executing instrumentation function invocations that manipulate different pieces of metadata simultaneously in different threads.

The key challenge to spreading instrumentation function execution across many threads is ensuring that metadata updates occur in the correct order, and do not suffer from data races. Metadata-based parallelization solves this problem by using a user-specified mapping of instrumentation function invocations to *serialization sets*. The runtime ensures that metadata updates are handled correctly by executing all function invocations in a given serialization set in the same thread. It achieves concurrency by spreading different serialization sets across multiple threads.

Metadata-based parallelization improves on previous techniques to reduce the overhead of program instrumentation of a broad class of dynamic monitoring tools, including those that measure common-case behavior, such as profilers, and those that check for anomalous behavior, such as debugging and testing tools. Our technique allows tool developers to leverage parallelism with a natural, intuitive programming interface, leaving the burden of correct synchronization of the parallelized execution to the instrumentation system.

We have modified the EEL instrumentation system to support metadata-based parallelization, and we evaluate our prototype by comparing the performance of parallelized instrumentation on both multicore and SMP systems. We show that the fast communication provided by the multicore system is a key enabler for fine-grained parallelization, achieving speedups averaging 4.3X for value profiling and 2.9X for data dependence profiling using 8 additional thread contexts.

1. INTRODUCTION

The advent of commodity multicore microprocessors has brought multiprocessing capabilities to mainstream computing. Because the multiple cores are integrated on a single die, they can provide extremely fast inter-processor communication. Thus multicore processors do not just provide multiprocessing for the masses, but actually enable new types of parallel applications that were not possible given the higher communication latencies of conventional symmetric multiprocessing (SMP) machines.

Parallelizing existing applications that were unable to benefit from the coarse parallelism of SMPs is one obvious way to leverage multicore processors. Another, perhaps less apparent, opportunity afforded by multicore processors is the ability to enhance applications to provide richer features, greater usability, and improved reliability. Program instrumentation—additional code inserted into a program to collect information about its runtime behavior—is such an enhancement. This information has many useful applications, such as profile-driven optimization, debugging and testing, and intrusion detection and reaction.

Program instrumentation inserts additional code and data into a program, and can incur substantial overheads. Tool builders face an inherent trade-off between the richness of runtime data collected, and the performance degradation incurred. Section 2 summarizes research aiming to alleviate these overheads. Historically, efforts to reduce instrumentation overhead have focused on reducing the frequency of instrumentation execution via sampling techniques [1, 2, 9, 23]. Sampling can significantly reduce the overheads of instrumentation, but sacrifices accuracy in the process. While this is often a desirable trade-off for tools which measure common-case behavior, sampling is unacceptable for tools that monitor uncommon events. Testing and debugging applications frequently fall within the class of tools which require *complete profiling*. For example, memory debuggers such as Memcheck [31] and Purify [24] would be far less useful if they only occasionally caught errors.

Parallelization is an attractive option for tools that require the entire program to be monitored, because the overall execution time can be reduced by overlapping work using multiple threads. Previous approaches to parallelizing program instrumentation have focused on minimizing the performance impact to the program. Because instrumentation code reads from, but rarely (or never) writes to program state, the instrumentation can be executed in a separate thread. While this decomposition allows the program to execute at nearly full speed, the instrumentation still executes very slowly. In many cases, the results of the instrumentation are what is interesting, so allowing the program to finish early is of little benefit.

This paper describes another type of parallelism present in instrumented code, which we call *metadata parallelism*. Metadata parallelism is the degree of independence among the operations performed by instrumentation code on its own, private data structures. Instrumented programs often contain large amounts of metadata parallelism, because metadata is stored about fine-grained program attributes such as memory addresses and program counter values. Previous approaches to parallel instrumentation have been unable to exploit this phenomenon because its fine granularity was not

amenable to the high communication latencies of SMPs. In this paper, we show that a metadata-based decomposition can leverage the fast communication provided by multicore processors to greatly reduce the overhead of program instrumentation.

Metadata-based parallelization, discussed in Section 3, modifies a conventional thread of execution (the *program thread*) to send the arguments of an instrumentation function to one of a number of *delegate threads*. A delegate thread, running on a separate hardware context, is responsible for receiving the arguments and executing the instrumentation function. The key challenge in metadata-based parallelization is ensuring that metadata dependences between the instrumentation functions are honored, maintaining the appearance of sequential execution. This achieved by mapping each invocation of an instrumentation function to a *serialization set*. The instrumentation system then guarantees that all function invocations within a given serialization set execute in the same order they would have in sequential execution.

Metadata-based parallelization of program instrumentation enables rapid development of fast, accurate profiling tools, without placing restrictions on the type of data collected, or placing onerous demands on the tool developer. The main limitation of this technique is that it is only beneficial when the overhead of an individual instrumentation function is greater than the overhead of passing its arguments to a delegate thread. Fortunately, the advent of multicore processors with fast inter-core communication allows for very fast communication of the instrumentation arguments.

We have implemented a prototype of metadata-based parallelization using the EEL instrumentation system [26]. We compare the performance of two profiling tools running on a multicore system vs. a conventional SMP system. Our results demonstrate that the faster inter-thread communication provided by CMPs enables parallelization of program instrumentation at a very fine granularity. Metadata-based parallelization speeds up value profiling [8] by an average of 4.3X and data dependence profiling [20] by an average of 2.9X using 8 additional thread contexts on an UltraSPARC T1-based Sun Fire T2000 system.

2. RELATED WORK

Leveraging the power of program instrumentation requires tool developers to deal with complex processor instruction sets and esoteric executable file formats. This problem is addressed by *instrumentation systems* [5–7, 10–12, 14, 15, 26, 27, 30, 32]. These systems present tool developers with abstractions such as routines, control-flow graphs, and simplified, machine-independent instruction representations. By hiding the details of executable formats and instruction set architectures, these systems facilitate the rapid development of correct, efficient, and portable tools for monitoring application execution. The programming model provided by metadata-based parallelization adheres to the philosophy of instrumentation systems by providing an intuitive interface for tool developers to exploit parallelism in their instrumented code.

The utility of program instrumentation has motivated researchers to examine ways to alleviate its overheads. One way to reduce these overheads is to reduce the frequency of execution of instrumentation. Ball and Larus have developed optimal algorithms for control flow profiling, which reduce the amount of instrumentation to the minimum amount needed to reconstruct a complete profile [3, 4]. *Convergent profiling* reduces the frequency of instrumentation execution once a profile converges to a steady-state [8]. Statistical sampling techniques [1,2,9,23] can significantly reduce the amount of times instrumentation must be executed. These techniques are orthogonal to parallelization, and they may be used in a complementary fashion.

Hardware support is an attractive option for inspecting the runtime properties of applications. Modern processors provide event counters for a wide variety of hardware performance metrics. Researchers have proposed more elaborate hardware support to collect more detailed data about running programs [13, 16, 17, 19, 22, 25, 34]. The advantages of hardware schemes is that they can achieve very low overhead, and often do not require modification of the original program. The disadvantage of hardware support is that it adds complexity to the design verification process. Metadata-based parallelization avoids this additional complexity by using general-purpose hardware to reduce the overhead of profiling.

Parallelization is an attractive option for hiding instrumentation overheads, due to the inherent parallelism between a program and instrumentation code. Patil and Fischer proposed executing the program in one thread, with a second thread executing a *shadow process* that checks the correctness of memory operations [29]. Shadow processing allows the original program to run at nearly full speed, but the instrumentation itself still suffers from very large overheads. The shadow process, which determines if there were errors in the execution, can experience slowdown by a factor of 10X or more. Metadata-based parallelization could be applied to the shadow process to greatly reduce this overhead.

Oplinger and Lam proposed executing instrumentation functions as speculative threads on a simultaneously-multithreaded (SMT) processor [28]. Their design does not explicitly exploit metadata parallelism, instead relying on thread-level speculation hardware to detect and rollback violations of metadata dependences. This means that instrumentation with dense metadata dependences may cause frequent violations, negating the benefits of parallelization. Furthermore, their scheme requires hardware support for thread-level speculation, which has not yet been implemented in mainstream microprocessors.

Shadow profiling [21] and SuperPin [33] are both enhancements to Pin [12] that allow it to perform parallel execution of instrumentation. Both of these systems modify a program to periodically fork off a shadow process (or timeslice, in the parlance of SuperPin) that executes a portion of the program instrumented with the desired data collection routines. These shadow processes are quite long–millions to hundreds of millions of instructions for shadow profiling, and up to a second for SuperPin. Because the only modification made to the original program is infrequent forking of a shadow process, the slowdown incurred by the program thread is minimal. However, shadow profiling is only applicable to event-counting forms of profiling, which have inherently parallel metadata updates (e.g. counter increments).

SuperPin provides an API for users to reconcile the local metadata updates of the individual timeslices. This API provides a merge function that is called at the end of each timeslice to integrate the local metadata updates with the overall metadata. When metadata updates depend on values of the metadata generated in previous timeslices, the tool developer must track all the updates in the local timeslice. Then when the merge function is called, the previous values of metadata are propogated through these updates to generate the final value. This may require significant extra bookkeeping

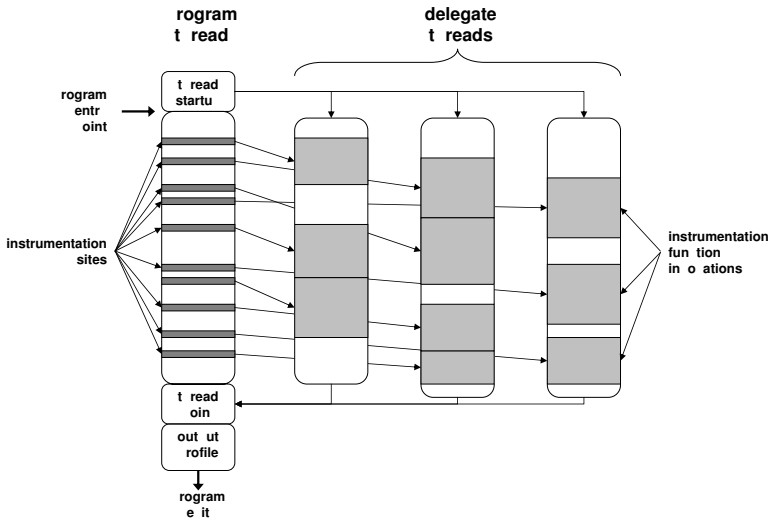

Figure 1: Program execution under metadata-based parallelization

effort, making it difficult to parallelize some types of instrumentation.

3. METADATA-BASED PARALLELIZATION

3.1 Overview

As an instrumented program executes, the instrumentation code maintains information in private data structures, or metadata. Metadata typically describe fine-grained aspects of the state of the running program. This leads to a great deal of independence in the operations performed by the instrumentation. For example, memory profiling tools track information about operations the program performs performs on memory by maintaining a metadatum for each distinct address. Updates to a particular metadatum depend only on its previous state, and the operation the program performs on the associated address. Updates to metadata for different addresses are thus completely independent.

We propose metadata-based parallelization to leverage this independence. The high level operation of this model is illustrated in Figure 1. The conventional thread of execution, or *program thread* initially forks a number of *delegate threads* before it begins execution. Then, as the program thread executes and encounters instrumentation sites, it sends the information needed by the instrumentation–the memory address and the operation performed on it, in our example–to a delegate thread, and continues execution. The delegate threads wait to receive this information, and when they do, they execute the instrumentation on behalf of the program thread. Metadata-based parallelization achieves concurrency by spreading the instrumentation over multiple delegate threads, speeding up the overall execution of the program.

There are two key challenges to metadata-based parallelization. First, instrumentation invocations that operate on the same metadatum must still execute in program order. This ensures that metadata operations correctly observe the previous value of the metadata, as is required by our memory profiling example. This is not guaranteed if instrumentation is distributed to delegate threads in an ad hoc fashion. Second, to prevent data races from occurring when different threads attempt to access the same metadatum, mutual exclusion to individual metadata elements must be ensured. Metadata-based parallelization solves these problems by executing all instrumentation that operates on the same metadatum in the same delegate thread. This solves the ordering problem, because delegate threads retrieve messages from the program thread in order, so updates to the same metadatum still happen in program order. The mutual exclusion property is also assured by executing all operations on a particular metadatum in the same thread. Restricting operations on individual metadata elements to a single thread does not overly restrict concurrency because there are typically many more metadata elements than there are threads.

3.2 Example: Data Dependence Profiling

We illustrate metadata-based parallelization with a specific example of memory profiling–data dependence profiling [20]. This technique is used to ascertain which static store instructions each static load instruction depends on during the dynamic execution of the program. This is achieved by shadowing memory with a table that tracks, for each byte in memory, the program counter of the last store to write to that address. Stores are instrumented to write their PCs into the table for each memory element they address. Loads are instrumented to access the shadow table for the addresses they access, and read out the PC of the last store to that address. This PC is then added to the list of stores the load depends on. The instrumentation functions for loads and stores are given in pseudocode in Figure 2.

```
ddp_store (addr st_pc, addr st_addr, int size) {
    // insert st_pc into shadow memory table
    // from st_addr to st_addr+size
}

ddp_load (addr ld_pc, addr ld_addr, int size) {
    // 1) access shadow memory table
    //    from ld_addr to ld_addr+size
    // 2) read store pcs, add to list of
    //    stores this load depends on
}
```

Figure 2: Data dependence profiler

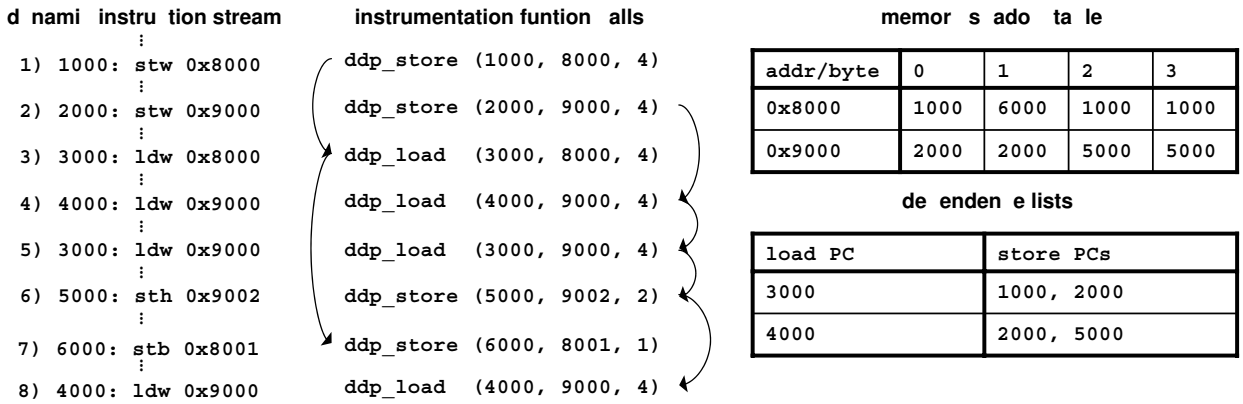

Figure 3: Data dependence profiling example

Figure 3 gives an example of the data dependence profiler monitoring a hypothetical instruction stream, showing the corresponding instrumentation function calls and the final state of the memory shadow table and the dependence lists. Metadata dependences between the instrumentation function invocations are indicated with arcs. For example, the load on line 3 depends on the store on line 1, because they are to the same address. Loads can also depend on multiple stores, because different sizes of loads and stores can overlap. This is the case with the dependence of the load on line 8, which depends on the store on line 6 and line 2. Anti-dependences also occur, such as the store on line 7 and the load on line line3.

For the data dependence profiler to correctly observe the memory behavior of the program, metadata dependences must be honored. Violation of these dependences might result in the load instrumentation determining that it depends on a store it does not actually depend on, or missing a store that it does depend on. On the other hand, instrumentation function invocations that are independent of each other may be ordered arbitrarily without changing their outcome. In our example, the invocations that access the `0x8000` row of the memory shadow table (the left dependence arcs) could be run in any order relative to the invocations that access the `0x9000` row of the memory shadow table (the right dependence arcs), and the same set of load dependence lists would result.

Metadata-based parallelization can be applied to this example by executing operations on the two different rows of the shadow table in different delegate threads. Thus the instrumentation functions invocations on lines 1, 3, and 7 would be executed in one thread, and the invocations on lines 2, 4, 5, 6, and 8 would be executed in another thread. The execution of these delegate threads can be arbitrarily interleaved without changing the results of the profiler.

3.3 Serialization Sets

Metadata-based parallelization ensures the necessary ordering between instrumentation function invocations using *serialization sets*. By placing invocations in the same serialization set, the tool developer indicates that they access the same metadatum. The runtime support of the instrumentation system then ensures that all function invocations in the same serialization set are assigned to the same delegate thread for processing. Note the distinction between functions and their invocations here–a single static function may have its dynamic invocations mapped to different serialization sets, and

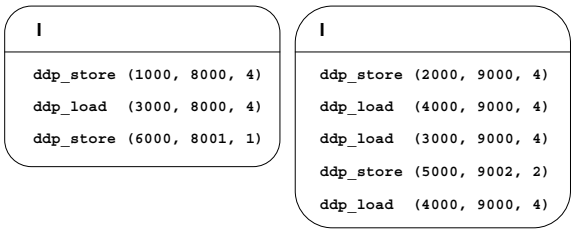

Figure 4: Serialization sets for data dependence profiling example

invocations of different functions may be mapped to the same serialization set.

The runtime mapping between instrumentation function invocations and serialization sets is performed using a user-specified *serialization mapping*, or *serializer*. The serializer is a piece of code that takes some or all of the arguments to the instrumentation function, and identifies the serialization set for the instrumentation function invocation. An extremely simple serializer could merely return the address of the metadatum manipulated by the function invocation. In practice, it is simpler to use the value of whatever piece of machine state the function invocation is related to. This approach provides a natural programming interface for writing instrumentation functions, and lends itself to efficient implementation, since these values are usually readily available at the instrumentation site. Serializers may be either static, such as the PC of a particular instruction, or dynamic, such as the memory address used by a load or store.

Serialization sets provide an intuitive programming interface. Writing instrumentation code already requires the tool developer to reason about the machine state the instrumentation is observing and to correctly track metadata related to that state. Specifying the serialization mapping requires the programmer to extend this reasoning to specify, for a given function invocation, which metadata element it will access.

Serializers are also easy to debug. Incorrectly specified serializers may be detected by placing a field in every metadata structure

that tracks the serialization set that accesses it. Then each instrumentation function invocation can check that its serialization set matches the value of this field, and signal an error on mismatch. Serializers that perform poorly are usually caused by load balancing issues, which may be discovered by observing the number of function invocations handled by each delegate thread. If a few delegate threads are performing most of the work, then the serializer could be modified to identify the metadata dependences at a finer granularity, or to provide a more uniform distribution of mappings to the serialization sets.

For our data dependence profiler example, a good serializer would be the address accessed by a memory instruction shifted left by two bits, which corresponds to the rows of the memory shadow table. Figure 4 shows the two serialization sets that result from our example using this serializer. This results in two sets of function calls that respect the dependence arcs shown in Figure 3. Thus while the members of each sets must run in the same delegate thread, these sets can be mapped to different delegate, yielding concurrent execution of the instrumentation functions.

3.4 Instrumentation Requirements

For the purpose of this paper, we assume all instrumentation is encapsulated in functions, and that all the state needed to execute the instrumentation function is passed in as arguments. This simplifies the expression of instrumentation by the tool builder, and is a common idiom in many instrumentation systems [5, 7, 11, 12, 14, 15, 26, 30, 32]. In practice, this is not a limitation, since most forms of instrumentation can be trivially transformed to meet this requirement.

We also assume that instrumentation functions do not change the original computation of the program. This is a fundamental requirement, since the implicit assumption in delegating an instrumentation function to another thread and proceeding with execution of the program is that the instrumentation function will return to where it was called.

We note that the underlying assumption of metadata-based parallelization—that instrumentation functions operate on a large number of distinct data elements—might seem to preclude the use of container data structures. Data structures such as arrays, lists, and hash tables can be important for achieving efficient implementations of algorithms. But if individual metadata elements are stored in a container, then accesses to the metadata elements, while independent, are still subject to races on the data structure containing them. A natural solution that leverages metadata parallelism is to partition the data structures per-thread, since metadata updates are confined to a single thread. For our data dependence example, the shadow table can be partitioned in this fashion, since all table accesses to a given metadatum only happen within a single delegate thread. It is possible that some instrumentation tools might require some programmer-specified synchronization on container structures, but we believe that metadata-based parallelization can handle most cases.

3.5 Examples

Table 3.5 lists examples of some common types of instrumentation tools, and how the serializer would be specified for each type. The first category, instruction profiling tools, collect information specific to individual instructions. Correct parallelization requires that all dynamic instances of a static instruction are seen in program order by instrumentation functions. With metadata-based parallelization, this is implemented by placing instrumentation for each instruction in a separate serialization set using a static serializer–the PC of the profiled instruction.

The second category contains memory profiling tools. These require that all updates to a given memory address are seen in the order they occur in the program. Because memory can be addressed at different granularities, instrumentation of memory instructions must be placed in serialization sets according to the largest granularity. Thus the serializer is the memory address with enough low-order bits masked off to ensure that updates of any size to a given address are assigned to the same serialization set. For example, on an architecture that can address up to eight bytes at a time, the serializer would be the address with the least-significant three bits shifted off.

4. IMPLEMENTATION

We have implemented a prototype of metadata-based parallelization using the EEL [26] instrumentation system. We chose to use EEL for three reasons: first, it is a proven instrumentation system used to develop several profiling tools [4, 26]; second, its source code is available, allowing us to add support for spawning and controlling delegate threads; and third, it supports the SPARC ISA, facilitating our comparison between a multicore system and a SMP system.

4.1 Programming Interface

Our implementation extends EEL's `call_snippet` interface to support parallel instrumentation calls with user-specified or built-in serializers as shown in Table 2. A `call_snippet` takes the specified instrumentation routine and generates code to pass arguments and invoke the routine. We extended this interface with a `pcall_snippet` which takes a user-supplied routine and serializer, and generates an *instrumentation proxy*, which is responsible for invoking the instrumentation function in a delegate thread. Users may write their own serializers as an `code_snippet` (EEL's generic form of instrumentation code), or use a built-in serializer. The built-in serializers are handy because they are simple to use, and have been optimized to execute efficiently.

Before instrumenting a program with `pcall_snippets`, tool builders must call the `prepare_parallel_instrumentation` method in EEL's executable class. This method adds the necessary symbols, routines, and library dependences to the application to correctly support the parallelized execution.

4.2 Runtime Support

Our parallel instrumentation system adds code to `main` function (or whichever function serves as the program entry point) to spawn the delegate threads on entry. It also adds code to perform a thread join at the end of this function, which ensures the application does not terminate before the delegate threads complete.

For each instrumentation site, the instrumentation system generates an instrumentation proxy. The instrumentation proxy performs 3 actions: 1) it executes the serializer to determine the serialization set; 2) it hashes the serialization set id over the number of delegate threads to identify the delegate for execution; and 3) it pushes the instrumentation function handle and arguments into the message queue.

Profiling Type	Examples	Serializer
Instruction	basic-block and edge profiling [3, 32], dynamic instruction counting, value profiling [8]	PC $>>$ $\log_2$ (instruction size)
Memory	memory debuggers [24, 31], data dependence profiling [20]	address $>>$ $\log_2$ (ACCESS_SIZE)

Table 1: Examples of profiling tools and how they would be serialized in metadata-based parallelization

Interface	Description
`call_snippet (routine* inst_r, call_args** args, bool returns_result);`	standard EEL call snippet (some parameters omitted for brevity)
`pcall_snippet (routine* inst_r, call_args** args, bool returns_result, code_snippet* serializer);`	parallel call snippet with user-specified serializer
`pcall_snippet_pc (routine* inst_r, call_args** args, bool returns_result, addr pc);`	parallel call snippet serialized on program counter `pc`
`pcall_snippet_target (routine* inst_r, call_args** args, bool returns_result, instruction* inst);`	parallel call snippet serialized on control-flow target of instruction `inst`
`pcall_snippet_address (routine* inst_r, call_args** args, bool returns_result, instruction* inst);`	parallel call snippet serialized on address generated by memory instruction `inst`
`pcall_snippet_value (routine* inst_r, call_args** args, bool returns_result, int_reg reg, addr_pc);`	parallel call snippet serialized on value in register `reg` before program counter `pc`

Table 2: Parallel instrumentation call interface

The instrumentation proxy uses a simple hash of the modulus of the serialization set id and the number of delegate threads. Since the modulus operation is fairly expensive in the SPARC ISA, we optimize the case where the number of delegate threads is of the form 2^N to generate the modulus by using the lower N bits rather than using the integer arithmetic instructions.

Communication between the program thread and the delegate threads is performed using one-way circular message queues. When the program thread executes an instrumentation proxy, the handle of the instrumentation function and all of its arguments are pushed into the appropriate message queue. A delegate thread then retrieves this message, first reading the handle of the instrumentation function. The delegate then uses the handle to look up how many arguments the function has, and then retrieves them from the message queue. When the program thread reaches the end of the `main` function in the program, it sends a special message with a null handle, which signals the delegate threads to terminate. The program thread then executes the thread join, which causes it to wait until all delegate threads finish processing instrumentation routines and exit.

4.3 Multithreaded Programs

Our current implementation supports only single-threaded applications, but there is no fundamental limitation that prevents metadata-based parallelization from being applied to multithreaded applications. This may be desirable if a multithreaded application executes with fewer threads than a machine has hardware contexts.

To support multithreaded execution, our tool would need to intercept and correctly handle process and thread control functions. Moseley et al. [21] describe how to handle the `fork` system call, which creates a new process. When `fork` is called and not immediately followed by an `exec`, new delegate threads must be created for this process. A parallelizing instrumentation tool must also catch all calls to all thread creation and termination routines to set up the necessary delegate threads.

5. EXPERIMENTAL EVALUATION

5.1 Experimental Setup

We evaluated the performance of our prototype implementation on both a multicore Sun Fire T2000 system, and an SMP Sun Fire V880 system. The configuration of these systems is described in Table 3.

The UltraSPARC-T1 processor [18] in the multicore system has eight processor cores, each of which are four-way multithreaded. When running parallelized instrumented programs on this machine, we bind the program thread to one core and do not map any other threads to that core. The delegate threads are then mapped to the

	Multicore Sun Fire T2000	SMP Sun Fire V880
Processor Type	UltraSPARC T-1	UltraSPARC-III+
# Processors	1	8
Cores per Processor	8	1
Threads per Core	4	1
Clock Speed	1GHz	900 MHz
Memory	16 GB	32 GB

Table 3: Machine parameters

Message Size (bytes)	Clock Cycles (Mean)	
	Multicore	SMP
4	45	49
8	45	63
16	45	92
32	45	129
64	45	243
128	45	294
256	45	310

Table 4: Mean number of clock cycles required to repeatedly send a message via the message queue

remaining cores in such a way as to minimize the number of threads sharing a core. This minimizes the amount of competition among delegate threads for processor resources.

Our benchmarks are comprised of all the C language benchmarks from the SPEC2000 benchmark suite. EEL is currently unable to correctly handle `eon` as well as the Fortran floating point benchmarks, so these are not included in our evaluation. The incompatibilities are a known issue with EEL, and are not related to our prototype implementation.

5.2 Communication Overhead

Our prototype implementation uses one-way message queues to communicate between the program thread and the delegate thread. The main limitation to performance of the prototype is how quickly the program thread can generate instrumentation arguments to pass to the delegate threads. This determines how much concurrency in the instrumentation can be exploited. To measure the rate at which the program thread is able to dispatch these messages, we wrote a microbenchmark where the program thread repeatedly sends messages of varying size to a delegate thread. Each message includes the tick counter of the processor running the program thread. By observing the deltas in the tick counter, we can measure the time needed to send each message. We used this microbenchmark to compare the overhead of sending message on the multicore and SMP systems. Table 4 summarizes the results.

The multicore system has a uniform overhead of 45 cycles, regardless of message size, due its a write-through L1 cache policy and shared on-chip L2 cache. The overhead in the SMP system is fairly similar for small messages, but increases quickly with increasing message size. By observing the individual stream of message latencies on the SMP system we found that each message that crosses a 64-byte cache line boundary incurred significant additional overhead, and that the effect was additive for messages that cross multiple cache line boundaries. The shared cache on the multicore system thus enables parallel applications with much greater amounts of communication.

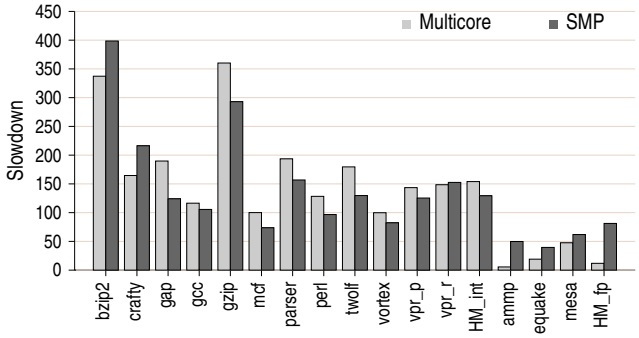

Figure 5: Overhead of value profiling

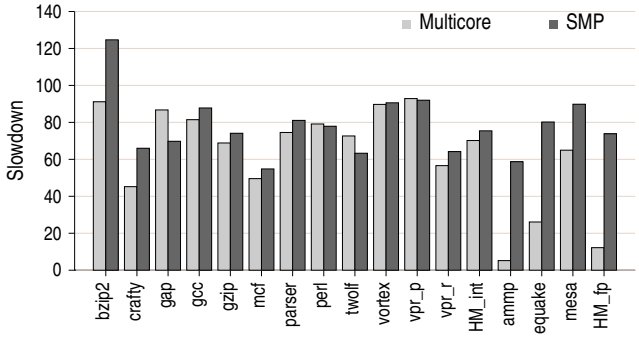

Figure 6: Overhead of data dependence profiling

5.3 Case Study 1: Value Profiling

For our first case study, we implemented value profiling [8]. Value profiling is used to observe the most common values produced by each instruction in a program. This information is valuable because it may be used to produce specialized versions of code when certain values in a program are invariant or very common.

Following the proposal of Calder et al [8], our implementation of value profiling maintains a 50-entry Top-N-Value (TNV) table for each instruction. The TNV table tracks the 50 most-commonly seen values, as well as the frequency of their occurrence. We measured the overheads of value profiling on both the multicore and SMP systems; the results are shown in Figure 5. Since the multicore and SMP systems use different CPUs, we compare their performance using the same set of instrumented programs to ensure that our parallelization experiments are addressing similar overheads.

Value profiling introduces significant overhead into program execution. For the integer benchmarks, the average average slowdown was 154X on the multicore system, and 129X on the SMP system. The slowdown for the floating point benchmarks was much lower on the multicore system (4X) than the SMP system (16X). This disparity reflects the very poor floating point performance of the UltraSPARC-T1 [18], which has limited floating point capabilities.

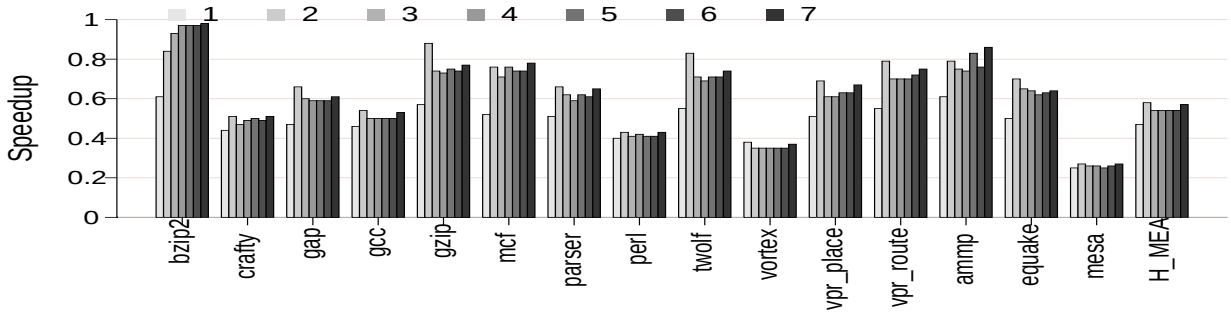

Figure 7: Performance of parallelized value profiling on an SMP system

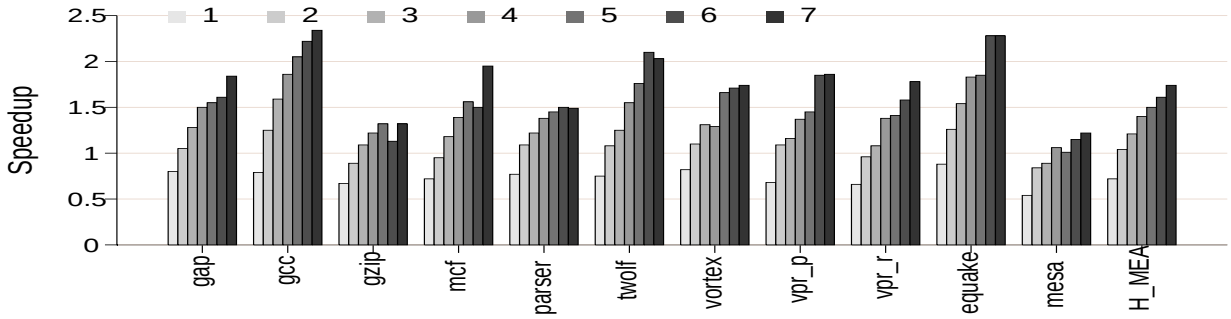

Figure 8: Performance of parallelized data dependence profiling on an SMP system

The floating point programs take much longer to execute on this system, so the instrumentation takes up a relatively smaller portion of the execution time.

To apply metadata-based parallelization to value profiling, we observe that the individual TNV table elements are each accessed only by the instrumentation for a particular instruction. Thus we parallelized the program using a PC-based serializer.

5.4 Case Study 2: Data Dependence Profiling

Our second case study was data dependence profiling [20], which has already been described in detail in Section 3.2. We parallelized this tool using the memory address based serializer described in the example. Two types of container data structures are used for this application—a hash table used to store the shadow memory table, and lists used to store the load dependences. We partitioned both into per-thread structures avoid needing to add any synchronization to the instrumentation. Since the load dependence lists are distributed over a number of thread-specific data structures, at the end of the profiling run we simply iterate over these results and combine them to obtain the final profile.

Figure 6 shows the overhead of data dependence profiling. The integer benchmarks had similar overheads of 70X and 75X on the multicore and SMP systems, respectively. The multicore system again showed less overhead in the floating point benchmarks, with an overhead of 12X compared to 74X for the SMP system.

5.5 SMP Performance

Figure 7 shows the performance of parallelized value profiling on the SMP system as a function of the number of delegate threads. Since value profiling instruments every instruction, a great deal of communication occurs between the program and delegate threads. As we saw in Section 5.2, the performance of communication degrades as the amount of data sent increases. The parallel value profiler suffers a slowdown by a factor of 2 for any number of delegate threads–the amount of communication in the parallelized value profiler is simply too much for the SMP to accommodate.

The performance of parallelized data dependence profiling on the SMP is given in Figure 8. Because data dependence profiling instruments only load and store instructions, the amount of communication is significantly less than in value profiling. This enables the parallelized profiler to achieve noticeable speedup. Parallel execution speeds up data dependence profiling by a factor of 1.75X for with 7 delegate threads.

5.6 Multicore Performance

The performance of parallelized value profiling and data dependence profiling on the multicore system is given in Figure 9 and Figure 10, respectively. These figures show the speedup achieved the multicore system using a varying number of delegate threads.

We use a variety of configurations to illustrate several factors affecting the performance. The first factor is the number of delegate threads mapped to a CPU. In the configurations using one to seven delegate threads, each delegate thread is mapped to a separate core. The 14 delegate thread uses exactly two threads per core. The second factor is that configurations using a number of delegate threads that is a power of two use a much faster hashing function in the

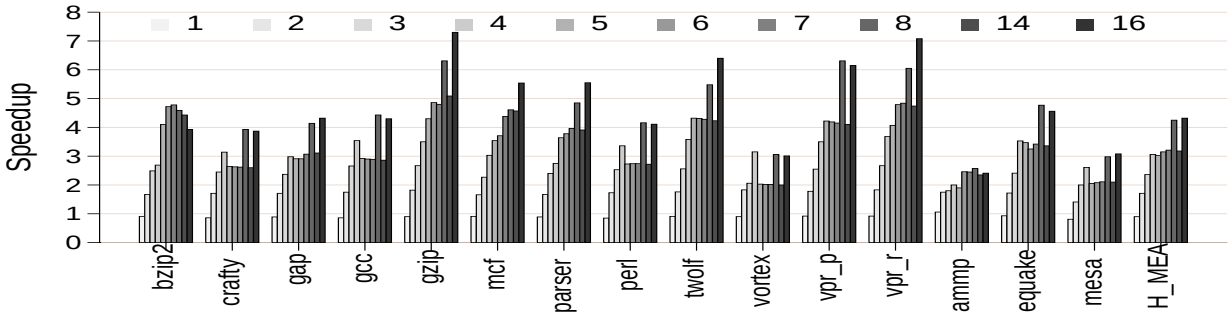

Figure 9: Performance of parallelized value profiling on a multicore system

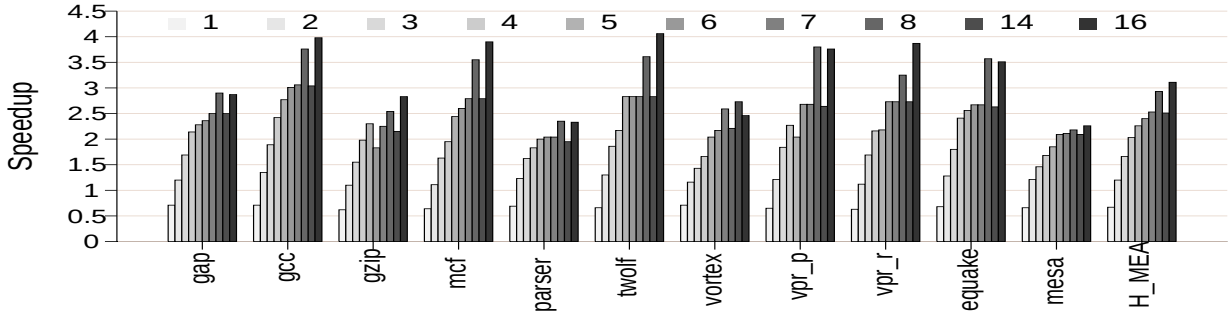

Figure 10: Performance of parallelized data dependence profiling on a multicore system

instrumentation proxy. This motivated our inclusion of 8- and 16-delegate thread configurations.

Using only one delegate thread results in a slight slowdown from the single-threaded benchmarks. The critical path in this configuration runs through the delegate thread. The slowdown indicates that reading arguments out of the message queue in the delegate thread before executing the instrumentation functions takes longer than executing the instructions between instrumentation functions in the single-threaded version.

The fast hash function used in the power-of-two configurations gives significant improvements over the configurations using the slower hash function. For example, the 4-delegate configuration runs faster than the 5-, 6-, and 7-delegate configurations for most benchmarks. And the 8-delegate configuration is much faster than the 7- or 14-delegate ones. The greatly improved performance of the faster hash function indicates that the speed of the instrumentation proxy plays a critical role in determining the amount of parallelism that can be exploited.

Comparing the 8-and 16-delegate configurations reveals that using multiple threads per core does not significantly improve performance for most of the benchmarks. This makes sense, because multithreading is intended to deal with the frequent L2 cache misses incurred by server workloads [18]. The SPEC benchmarks represent common desktop and workstation applications that miss far less often in the L2 cache.

Overall, metadata-baed parallelization achieves significant speedups utilizing the fast communication provided by the multicore system. Value profiling speeds up by a factor of 4.3X, and data dependence profiling speeds up by a factor of 2.9X, when using 8 delegate threads.

6. CONCLUSION

In this paper, we observe that program instrumentation typically collects numerous independent pieces of information (metadata) about a program's execution. We propose *metadata-based parallelization* to leverage this parallelism to greatly reduce the overhead of instrumented programs.

The main challege to parallelizing instrumentation is ensuring that metadata dependences are respected during the concurrent execution. We address this problem with *serialization sets*, a programming abstraction that allows tool developers to specify on what metadatum a particular invocation of an instrumentation function will operate. The runtime support in the instrumentation system then ensures that metadata dependences are respected by running all function invocations in a serialization set in the same thread. Serialization sets provide an intuitive programming interface, allowing tool developers to easily exploit fine-grained parallelism in a wide range of program-monitoring applications. Looking forward, we believe that this natural programming idiom can be useful for many other types of applications.

We have developed a prototype implementation of metadata-based parallelism within the EEL instrumentation system. We show that the fast on-chip communication provided by multicore processors is critical to enabling fine-grained parallelization. Using a multicore system, we then demonstrate that our prototype can reduce the overhead of two profiling applications, speeding up value profiling

by 4.3X and data dependence profiling by 2.9X using 8 additional hardware contexts.

Acknowledgments

We thank William Benton and Nicholas Kidd for feedback on this report. This work is supported in part by National Science Foundation (NSF) grants CCF-0702313 and CNS-0551401, funds from the John P. Morgridge Chair in Computer Sciences and the University of Wisconsin Graduate School. Sohi has a significant financial interest in Sun Microsystems. The views expressed herein are not necessarily those of the NSF, Sun Microsystems or the University of Wisconsin.

7. REFERENCES

[1] J. M. Anderson and et al. Continuous profiling: where have all the cycles gone? In *Proc. SOSP*, pages 1–14, 1997.

[2] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *Proc. PLDI*, pages 168–179, 2001.

[3] T. Ball and J. R. Larus. Optimally profiling and tracing programs. In *Proc. POPL '92*, pages 59–70, 1992.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *Proc. MICRO*, pages 46–57, 1996.

[5] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, September 2004.

[6] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proc. SIGMETRICS*, pages 128–137, 1994.

[7] A. S. et al. Vulcan: Binary tranformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Corporation, 2001.

[8] B. C. et al. Value profiling. In *Proc. MICRO*, pages 259–269, 1997.

[9] B. L. et al. Bug isolation via remote program sampling. In *Proc. PLDI*, pages 141–154, 2003.

[10] B. M. C. et al. Dynamic instrumentation of production systems. In *Proc. USENIX Annual Technical Conference*, 2004.

[11] B. P. M. et al. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.

[12] C. L. et al. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI*, pages 190–200, 2005.

[13] J. D. et al. Profileme: hardware support for instruction-level profiling on out-of-order processors. In *Proc. MICRO*, pages 292–302, 1997.

[14] J. M. et al. Diota: Dynamic instrumentation, optimization, and transformation of applications. In *Proc. WBT*, 2002.

[15] K. S. et al. Retargetable and reconfigurable software dynamic translation. In *Proc. CGO*, pages 36–47, 2003.

[16] K. V. et al. A programmable hardware path profiler. In *Proc. CGO*, pages 217–228, 2005.

[17] M. C. M. et al. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proc. ISCA*, pages 136–147, 1999.

[18] P. K. et al. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[19] S. S. S. et al. Rapid profiling via stratified sampling. In *Proc. ISCA*, pages 278–289, 2001.

[20] T. C. et al. Data dependence profiling for speculatie optimizations. In *Proc. CC*, pages 57–62, 2004.

[21] T. M. et al. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proc. CGO*, pages 198–208, 2007.

[22] T. M. C. et al. Using branch handling hardware to support profile-driven optimization. In *Proc. MICRO*, pages 12–21, 1994.

[23] X. Z. et al. System support for automatic profiling and optimization. In *Proc. SOSP*, pages 15–26, 1997.

[24] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. Winter USENIX Conf.*, pages 125–136, 1992.

[25] T. Heil and J. E. Smith. Relational profiling: enabling thread-level parallelism in virtual machines. In *Proc. MICRO*, pages 281–290, 2000.

[26] J. R. Larus and E. Schnarr. Eel: machine-independent executable editing. In *Proc. PLDI*, pages 291–300, 1995.

[27] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. PLDI*, pages 89–100, 2007.

[28] J. Oplinger and M. S. Lam. Enhancing software reliability with speculative threads. In *Proc. ASPLOS*, pages 184–196, 2002.

[29] H. Patil and C. N. Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *Software–Practice and Experience*, 27(27):87–110, 1997.

[30] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, and B. Bershad. Instrumentation and optimization of win32/intel executables using etch. In *NT '97: Proceedings of the USENIX Windows NT Workshop*, pages 1–7, 1997.

[31] J. Seward and N. Nethercote. Using valgrind to detect uindefined value errors with bit-precision. In *Proc. USENIX Annual Technical Conference*, 2005.

[32] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. In *Proc. PLDI*, pages 196–205, 1994.

[33] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proc. CGO*, pages 209–220, 2007.

[34] C. B. Zilles and G. S. Sohi. A programmable co-processor for profiling. In *Proc. HPCA*, page 241, 2001.