A STUDY OF THE VALIDITY AND UTILITY OF

PAPI PERFORMANCE COUNTER DATA

LEONARDO SALAYANDIA

Computer Science Department

APPROVED:

Patricia J. Teller, Ph.D., Chair

Steven M. Roach, Ph.D.

David H. Williams, Ph.D.

Charles H. Ambler, Ph.D.
Dean of the Graduate School

A STUDY OF THE VALIDITY AND UTILITY OF PAPI PERFORMANCE COUNTER DATA

by

LEONARDO SALAYANDIA, B.S.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Computer Science Department

THE UNIVERSITY OF TEXAS AT EL PASO

December 2002

ACKNOWLEDGEMENTS

I would like to acknowledge my advisor, Dr. Patricia J. Teller and my committee members, Dr. Steven M. Roach and Dr. David H. Williams for their time and support in making the completion of this thesis possible.

I would also like to thank my student colleagues for making my graduate education experience a memorable one. Your collaboration and friendship has been the perfect blend that makes the line between work and play invisible.

I would like to give special thanks to Dr. Ann Q. Gates and Dr. Patricia J. Teller for being such wonderful mentors since my undergraduate studies. Your encouragement, guidance and support were indispensable for helping me overcome yet another hurdle.

Finally, I would like to thank my family for their unlimited support. Mamá, papá, hermano, gracias por todo su cariño y apoyo.

ABSTRACT

Today, most modern microprocessors include monitoring hardware in the form of on-chip counters that can be used to obtain data about the performance of the microarchitecture and memory hierarchy (i.e., the various levels of cache and the translation-lookaside buffer, TLB). Initially, these counters were meant for the use of computer architects and operating system developers and, on most platforms, were not readily accessible to the general application developer. The Performance API (PAPI) cross-platform hardware counter interface facilitates programmer access to this information, which is delivered as event counts (e.g., the number of executed load instructions). Potentially, this data can be useful for the performance tuning of applications. However, in some cases, limited knowledge of the microarchitecture's structure and management algorithms may reduce the usefulness of event counts. Also, because of the overhead introduced by PAPI and/or the counting method implemented (i.e., aggregate or sampling), the information obtained from the counters may not capture the accuracy that is needed for a specific use of the data. Additionally, the PAPI interface, itself, or the hardware implementation may, inadvertently, introduce errors.

To address these issues and, thus, allow the programmer to use event count data with confidence, research is needed to determine when and how event count data can be used. To ascertain this information, a set of validation microbenchmarks, which stresses the platform in predictable ways, has been designed and developed. These benchmarks permit prediction of event counts and, thus, evaluation of event count data. Depending on

the platform and event under study, a complementary configuration microbenchmark may need to be designed and developed as well. The purpose of a configuration-microbenchmark is to attain information about when and how a specific event is generated. In some cases, this information is needed to design and develop a validation microbenchmark.

Several of these validation and configuration microbenchmarks are a product of the research reported in this thesis. They are associated with the following events:

- data TLB misses,
- L1 data cache (Dcache) misses,
- L2 Dcache misses,
- cache intervention requests,
- cache invalidation requests,
- requests for exclusive access to a shared cache line, and
- requests for exclusive access to a clean cache line.

The behavior of these events was studied on the following platforms: SGI MIPS R10000, IBM Power3 and Intel Itanium. The main results of this study, which compares predicted and hardware-reported event counts, indicate that (1) for some of the events studied, the use of validation microbenchmarks can only demonstrate “reasonableness” of hardware-reported counts, in contrast to using them to validate the counts, and (2) the usefulness of event count data (to a programmer) is dependent upon both the nature of the application and the type of event monitored.

TABLE OF CONTENTS

Page

Abstract iv

List of Tables ix

List of Figures x

Chapter

1. INTRODUCTION 1

2. METHODOLOGY 5

    2.1 Microbenchmarks 5

    2.2 Data collection 8

    2.3 Predicted vs. hardware-reported event counts 9

3. STUDIED PLATFORMS 11

    3.1 Platform A: MIPS R10000 (SGI Origin 2000) 11

    3.2 Platform B: IBM Power3 14

    3.3 Platform C: Intel Itanium 14

4. TARGET EVENTS 15

    4.1 Data TLB miss event 18

        4.1.1 Validation microbenchmark 19

        4.1.2 Acquiring parameters for the validation microbenchmark 21

        4.1.3 Data collection 26

        4.1.4 Predicted vs. hardware-reported event counts 26

4.2 L1 data cache miss event 28
4.2.1 Validation microbenchmark..... 29
4.2.2 Data Collection 33
4.2.3 Predicted vs. hardware-reported event counts 33
4.3 L2 data cache miss event 36
4.3.1 Predicted vs. hardware-reported event counts 37
4.4 Cache intervention request event 38
4.4.1 Validation microbenchmark..... 39
4.4.2 Data collection 41
4.4.3 Predicted vs. hardware-reported event counts 42
4.5 Cache invalidation request event 43
4.5.1 Validation microbenchmark..... 45
4.5.2 Data collection 47
4.5.3 Predicted vs. hardware-reported event counts 48
4.6 Request for exclusive access to a shared cache line 51
4.6.1 Validation microbenchmark..... 52
4.6.2 Data Collection 54
4.6.3 Predicted vs. hardware-reported event counts 55
4.7 Request for exclusive access to a clean cache line 56
4.7.1 Validation microbenchmark..... 57
4.7.2 Data collection 60
4.7.3 Predicted vs. hardware-reported event counts 60

5. CONCLUSIONS AND FUTURE WORK 62

5.1 Summary of results 62

5.2 Future work 64

References 65

Appendix A: PAPI instrumentation code 67

Appendix B: Hardware-reported data 69

Appendix C: benchmark code 77

Curriculum Vitae 101

LIST OF TABLES

Page

Table 1. Native R10000 instructions for shared-memory parallel programming. 13

Table 2. Description of PAPI events under study. 15

Table 3. Summary of events studied. 63

LIST OF FIGURES

Page

Figure 1. Predicted vs. hardware-reported count comparison formula. 10

Figure 2. Configuration of the SGI Origin 2000 multiprocessor. 12

Figure 3. DTLB miss benchmark algorithm. 20

Figure 4. Saavedra's benchmark results for the R10000 and the Itanium platforms. .. 22

Figure 5. Storage of a process in memory. 23

Figure 6. Distribution of variables in the data segment. 24

Figure 7. Padding benchmark algorithm. 25

Figure 8. Padding benchmark results for 100 iterations. 26

Figure 9. DTLB miss validation benchmark results..... 27

Figure 10. L1 Dcache miss benchmark algorithm. 30

Figure 11. Shared-memory L1 Dcache miss benchmark algorithm. 32

Figure 12. L1 Dcache miss validation benchmark results..... 34

Figure 13. L2 Dcache miss validation benchmark results..... 37

Figure 14. Cache intervention requests benchmark algorithm. 40

Figure 15. Cache intervention requests validation benchmark results. 42

Figure 16. Cache invalidation requests benchmark algorithm. 47

Figure 17. Cache invalidation requests validation benchmark results. 48

Figure 18. Branch mispredictions on the cache invalidation requests benchmark. 50

Figure 19. Requests for exclusive access to a shared cache line benchmark algorithm. 53

Figure 20. Requests for exclusive access to a shared cache line validation benchmark results. 55

Figure 21. Requests for exclusive access to a clean cache line benchmark algorithm. 58

Figure 22. Requests for exclusive access to a shared cache line validation benchmark results. 61

Chapter 1

INTRODUCTION

Most modern microprocessors contain self-monitoring hardware that can be used to obtain data about the performance of the microarchitecture and memory hierarchy (e.g., the various levels of caches and translation lookaside buffers, TLBs). This monitoring hardware, which mainly is in the form of on-chip counters, is designed to provide insight into the behavior of a microprocessor while executing an application, but without affecting its performance. Initially, these counters were meant for the use of computer architects and operating system developers and, on most platforms, were not readily accessible to the general application developer. Nonetheless, some application and compiler developers gained access to them and found that the type of information provided by the counters could guide them through the process of fine-tuning application performance and provide feedback to optimizing compilers that could potentially improve code scheduling/pipelining **[1]**.

The level of availability and access to the counters varies from platform to platform. It usually is difficult and time-consuming for the average application developer to use the counters because it requires him/her to have in-depth knowledge about low-level processor details. The Performance API (PAPI) project was initiated at the University of Tennessee at Knoxville with the purpose of providing a cross-platform API that defines a standard set of events and maps as many of these as possible to each platform. The semantics of the events is necessarily platform dependent but the purpose of PAPI is not

to provide a standard definition. Instead, it attempts to provide a set of events that is considered by the performance computing community to be the most important for performance analysis purposes **[1]**.

PAPI has been used successfully by many application developers and is now incorporated in several comprehensive performance tools **[2]**. For example, the *perfometer* tool, also developed by the PAPI research team, provides a graphical view of performance counter data as it is being generated. The interface (a cross-platform Java front-end) allows the user to change the event that is being monitored and has the ability to monitor parallel applications. Instrumentation of the monitored application is done by inserting a function call at the beginning of the code. The user can then monitor a variety of events without having to change the code. Other tools developed by Sandia National Laboratory, The University of Illinois, and Pacific-Sierra Research, among others, also have incorporated PAPI support into their performance analysis tools **[2]**.

The wide acceptance of PAPI as a performance tool emphasizes the importance of evaluating the accuracy and usefulness of event count data collected via PAPI, issues addressed by this thesis and the research reported on in **[3]**, **[4]**, and **[5]**. PAPI library calls, which are inserted in the monitored program, may perturb the reported counts (e.g., due to the execution of instructions that set and read the counters). Also, it is possible that the counters may be implemented incorrectly or the reported counts may be inexplicable without in-depth knowledge of the idiosyncrasies of the microarchitecture. Information that sheds light on these issues is of import to application programmers who are using

event count information to tune the performance of their programs. This is the prime motivation for the Department of Defense's support of this work.

Previous work has shown that the hardware-reported counter data may not be accurate when the granularity of the measured code is insufficient to ensure that the overhead introduced by the counter interface does not dominate the event count **[5]**. Also, previous evaluation of performance counter data across platforms has resulted in error characterization. Through the use of benchmarks that are specifically designed to stress the microarchitecture in predictable ways, performance counter data is analyzed by comparing expected counts with hardware-reported counts. This comparison led to the establishment of the following error categories proposed in **[3]**:

1. overhead or bias,
2. multiplicative,
3. random, and
4. unknown.

An overhead or bias error refers to a constant difference observed between expected and hardware-reported counts for a given event measured using versions of a benchmark that differ in terms of the number of times the event is generated. A multiplicative error refers to the case when hardware-reported counts exceed expected counts by a defined factor. A random error occurs when expected counts and reported counts differ significantly, but only part of the time. Finally, an unknown error happens when there is no apparent relationship between the expected and hardware-reported counts; this may be due to either a combination of the previous types of errors or unidentified processor

behavior. The work presented in this thesis is, in part, reported on in **[3]** and is, in part, an extension of that work. The following seven events are the focus of this thesis:

1. data TLB (DTLB) misses,
2. L1 data cache (Dcache) misses,
3. L2 Dcache misses,
4. cache intervention requests,
5. cache invalidation requests,
6. requests for exclusive access to a shared cache line, and
7. requests for exclusive access to a clean cache line.

This work evaluates the accuracy of the hardware-reported counts associated with these events and/or indicates under what circumstances the data can be used. These events were chosen because of their association with the performance of the memory hierarchy, which usually defines the critical path in the overall performance of an application. The last four events are related to the performance of shared-memory multiprocessors that ensure cache coherence. The second and third events, previously studied by **[6]** on a uniprocessor, are addressed again on a multiprocessor platform.

The general methodology used to study hardware-reported event counts is discussed in Chapter 2 and the targeted platforms are described in Chapter 3. Chapter 4 discusses the events under study, and presents and analyzes the results of the study. Chapter 5 contains concluding remarks and a description of future work.

Chapter 2

METHODOLOGY

The method used to evaluate the data reported by performance counters, called the *validation process*, is similar to that used in **[3]**. The process consists of seven phases, which are repeated as necessary. For each event under study, the phases are as follows.

1. **Microbenchmark**: design and implement a validation microbenchmark that permits event count prediction.
2. **Prediction**: predict event count using a mathematical model.
3. **Data collection-1**: collect (hardware-reported) event count data using PAPI.
4. **Data collection-2**: collect event count data using a simulator (not always necessary or possible).
5. **Comparison**: compare predicted and collected event counts.
6. **Analysis**: analyze results to identify and possibly quantify differences.
7. **Alternate approach**: when analysis indicates that prediction is not possible, use an alternate means to either verify reported event count accuracy or demonstrate that the reported event count seems reasonable.

2.1 Microbenchmarks

The focal point of this methodology relates to phase 1, the design and development of the *validation microbenchmark*. A validation microbenchmark is a program that is small in size and has an execution pattern that is easily traceable. The purpose of the program is to stress the platform in predictable ways with respect to the event under

study. In designing a microbenchmark, one needs to consider both the structure and configuration of the microarchitecture and memory hierarchy. Whenever possible, the requisite microarchitecture characteristics are obtained from platform documentation. When this is not possible, a *configuration microbenchmark* is developed to help deduce the missing details. Such is the case for the data TLB miss event discussed in Section 4.1, where a configuration microbenchmark is used to deduce the user data page size needed for the validation microbenchmark.

The validation microbenchmark design also needs to take into account the definition of the PAPI event under study. As mentioned before, the purpose of PAPI is to provide general descriptions of the cross-platform events rather than giving the exact semantics for them. Similarly, validation microbenchmarks are designed to apply across the platforms of interest. However, the analysis of the results obtained from executing validation benchmarks needs to consider the details of the experimental platform. These cross-platform validation microbenchmarks are developed in the ANSI C language and can be thought of as generic benchmarks that can be cross-compiled to execute on a specific target platform. At this point, the benchmarks do not include platform-dependent system calls (e.g., system calls to handle process synchronization) and in some cases customization of the generic microbenchmark may be needed before it can be used to validate an event on a specific platform. The generic version of a microbenchmark also serves as a template from which a test suite is generated. A *test suite* is comprised of a set of different versions of a generic microbenchmark. The versions differ with respect to the number of events that are expected to generate. As demonstrated in **[5]**, such a suite can

be used to determine the overhead introduced by the counter interface. Using this information, an equation can be defined to quantify the portion of an event count that is attributable to the counter interface. A test suite also is used to validate assumptions made with respect to expected event counts or to provide insight that may lead to modification of such assumptions. For example, the generic version of the cache invalidation requests microbenchmark, which deals with a multiprocessor environment (discussed in Section **4.5**), is customized to execute on the MIPS R10000 platform by adding native system calls that assign multiple processes to execute on separate processors. Also, manipulating the size of the generic benchmark's main for-loop, which determines the number of events generated by the benchmark, produces multiple versions of the benchmark.

Previous work **[3]** related to performance counter data evaluation has identified the following classes of validation microbenchmarks that are used to study a variety of events:

1. array microbenchmark,
2. loop microbenchmark,
3. in-line microbenchmark, and
4. floating-point microbenchmark.

The array microbenchmark consists of code that traverses an array at defined strides with the purpose of stressing the data portions of the various levels of the memory hierarchy and allows prediction of related events, e.g., L1 Dcache misses. The loop microbenchmark, which consists of a sequence of instructions within a loop, is used to stress a particular functional unit. For example, it was used to study the number of stores

executed. The in-line microbenchmark is the unrolled version of the loop microbenchmark, and its purpose is to stress the instruction portions of the various levels of the memory hierarchy. Events such as L1 Icache misses are its target. Finally, the floating-point microbenchmark is a variation of the loop microbenchmark where the sequence of instructions is replaced by floating-point instructions; it has the purpose of stressing the floating-point units. Events like floating-point add operations executed are studied using this benchmark.

Since the events under study in this thesis deal primarily with the data portion of the memory hierarchy, only versions of the array microbenchmark are used. Furthermore, a new class of microbenchmark is presented that permits the study of memory hierarchy related events in a multiprocessor environment. This is called a *ping-pong microbenchmark* and consists of code that forces multiple processes to alternately access one or more shared variables. The purpose is to stress the cache coherency unit. This benchmark is discussed in Section **4.5**.

2.2 Data collection

In order to study an event, the validation microbenchmark is instrumented by inserting initialization code that sets up PAPI to monitor the specific target event. Also, PAPI calls to start and stop the counters are inserted to delineate that portion of the benchmark that is under study. For example, the area of interest may exclude initialization code or code that does not stress the event under study in an easily traceable manner. The generic validation microbenchmark designed to stress the part of the microarchitecture or memory hierarchy associated with the event under study is

customized to generate an expected count, also called a *predicted count*. Each version of the microbenchmark is considered a *test case*, where each test case is identified by its predicted count. A test case is executed 100 times to get an *average test-case count*, which is used in the analysis phase of the validation process. Using an average takes into account the variability of the reported counts. In order to monitor the stability of the average test-case count, the standard deviation of each of the 100 instances of a test case is computed and test cases with large standard deviations are the targets of further study. A script, as opposed to wrapping the microbenchmark in a for-loop, is used to run a sequence of 100 instances of a test case. Wrapping a validation microbenchmark in a for-loop would cause reuse of benchmark and PAPI data as well as instructions and, consequently, could eliminate or introduce some events that otherwise would not be. Finally, a *test suite*, which is a collection of test cases for a single event, is used to study the behavior of the platform as the granularity of the test case increases, i.e., as the predicted number of events generated by the sequence of test cases increases. In general, the test suites are comprised of test cases that are expected to generate from 1 to 1,000,000 instances of the event under study. For some events, due to platform limitations or benchmark design, a smaller test suite is used.

2.3 Predicted vs. hardware-reported event counts

Once the data collection phase has been completed, the average tests-case counts are compared against the predicted counts. A percentage difference is computed for such a purpose using the formula given in Figure 1. A positive percentage difference indicates that the hardware-reported counts tend to be larger than the predicted counts. A negative

percentage difference indicates the opposite. The absolute value of the percentage difference is used to graph the results.

% difference = (Reported Count – Predicted Count) / Predicted Count

Figure 1. Predicted vs. hardware-reported count comparison formula.

Analyzing the percentage difference across a test suite is necessary to categorize the error. As mentioned in the introductory section, the error categories proposed by **[4]** are used. Understanding the nature of the error is essential to providing information that might explain its source and, thus, aid application developers in using the event count to tune the performance of their codes. For example, a constant difference of zero is the ideal situation, where the expected counts are exactly the same as the hardware-reported counts. This situation would indicate that the counter interface does not introduce overhead to the count, the hardware is monitoring what is expected, and the hardware is behaving as is expected. On the other hand, a percentage difference that starts out large for small test cases and approaches zero as the test cases get larger indicates that there is an “overhead” or “bias” type of error, which may possibly be due to the PAPI interface.

Chapter 3

STUDIED PLATFORMS

Three platforms were used in this study:

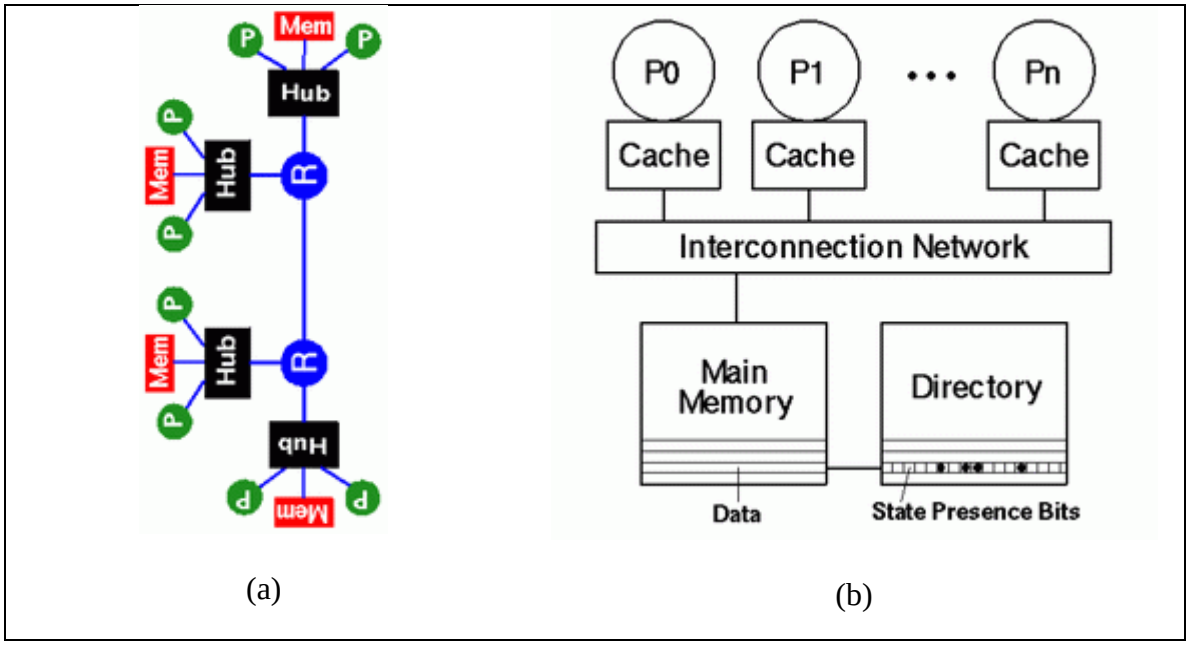
1. SGI's MIPS R10000,
2. IBM's Power3, and
3. Intel's Itanium.

These platforms were chosen because of their interest to the Department of Defense, which indirectly funds this research. Not all of these platforms support all the events under study. All three platforms where used to study the DTLB miss event discussed in Section 4.1. Only the MIPS R10000 platform was used for the other six events under study; this is because of the local availability of a SGI Origin 2000, the processors of which are R10000s. Both the Power3 platform and the Itanium platform used in this study are uniprocessor machines. The architectural characteristics of each platform, applicable to the events under study, are given in the following sections as described in their respective documentation **[7]**, **[8]**, **[9]**, **[10]** and **[11]**.

3.1 Platform A: MIPS R10000 (SGI Origin 2000)

The SGI Origin 2000 multiprocessor machine used in this study is comprised of eight MIPS R10000 processors, revision 2.6, each of which runs at 180 MHz. This processor has a two-way set associative, 32K byte (32KB) L1 data cache (Dcache) with a 32-byte line size and a two-way set associative, 32KB L1 instruction cache (Icache) with a 64-byte line size. The 1MB L2 cache is unified and is off chip; it is two-way set

associative and its line size is 128 bytes. Both the L1 and L2 caches have a least recently used (LRU) replacement policy. The TLB, which is suspected to be unified, has 64 entries and is fully associative. Page size support ranges from 4KB to 16MB, increasing in powers of 4. The platform implements a directory-based cache coherency protocol, where coherency is kept at the L2 cache level (i.e., inclusion is maintained between the L1 and L2 caches of each processor). There are two processors per node and the nodes are connected through routers. The 2GB of main memory is uniformly distributed across nodes. Figure 2 shows the general configuration of the platform as well as the hardware used to implement the directory-based protocol.



(a) Shows the distribution of processors and memory in the SGI Origin 2000 multiprocessor. Each node contains two processors with 512MB of main memory. The processors are connected through hubs and the hubs are interconnected through routers. (b) Illustrates the hardware used to implement Origin 2000's directory-based cache coherency protocol.

Figure 2. Configuration of the SGI Origin 2000 multiprocessor.

The operating system for this platform is the IRIX 6.5; it supports the set of system calls described in Table 1 that facilitate shared-memory parallel programming. These system calls are used in the microbenchmarks and are associated with events (i.e., all the events studied in this thesis except data TLB misses) that occur in a shared-memory multiprocessor environment. The benchmarks were compiled by the gcc compiler, version 2.95.2, release 19991024. Finally, the platform includes two 32-bit performance counters. Each counter can monitor one event at a time and there is a choice of 16 different events for each. Refer to **[8]** for a list and description of each.

Table 1. Native R10000 instructions for shared-memory parallel programming.

Instruction	Description
m_set_procs	Sets the number of processes to be forked when m_fork() is called. Parent process is included in count.
m_fork	Creates n-1 processes that execute a function in parallel with the calling process.
m_sync	Synchronizes all executing threads at the m_sync-defined point in the code. A thread busy waits until all other threads call the m_sync function, at which point all threads resume after the m_sync call.
m_get_myid	Returns the thread identifier, which ranges from zero to n-1.
sysmp (MP_MUSTRUN, proc_num)	Provides control and/or information for miscellaneous system services. The MP_MUSTRUN command assigns the calling process to execute on the processor specified by proc_num.

3.2 Platform B: IBM Power3

This platform is a uniprocessor that uses an IBM Power3 running at 200 MHz. The processor has a 128-way set-associative, 64KB L1 Dcache with a 128-byte line size and a 128-way set associative, 32KB L1 Icache with a128-byte line size. The L2 cache is off chip; it is a direct-mapped 4MB cache with 128-byte line size. The TLB, which is suspected to be unified, is 2-way set associative with 256 entries and an LRU replacement policy. One page size, 4 KB, is supported. The operating system is AIX 4.3. The microbenchmarks were compiled using gcc version 2.7.2.3. The processor includes eight counters that support over 100 events.

3.3 Platform C: Intel Itanium

This platform is a uniprocessor that uses an Intel Itanium processor running at 733 MHz. The processor has a 4-way set-associative, 16KB L1 Dcache with a 32-byte line size and an L1 Icache with the same configuration. The 96KB L2 cache is unified; it is 6-way set-associative with a line size of 64 bytes. This platform contains a fully-associative ITLB with 64 entries and two levels of DTLBs, both fully-associative. The L1 DTLB has 32 entries and the L2 DTLB has 96 entries. The supported page sizes are: 4 KB, 16 KB, 64 KB, 256 KB, 1 MB, 4 MB, 16 MB, 64 MB, and 256 MB. The operating system is Linux 2.4.18. The benchmarks were compiled using gcc version 2.96, release 20000731 for Red Hat Linux 7.1 2.96-101. The platform includes four 32-bit counters that support 150 events. Refer to **[10]** for details about the events supported.

Chapter 4

TARGET EVENTS

The events under study, as described by the PAPI website **[12]**, are shown in Table 2. The table denotes the PAPI-assigned name for each event, its PAPI-assigned number, and the description of the event. The PAPI event name or its number is used to setup PAPI to monitor the event. Appendix A shows the PAPI setup code that was used to instrument the validation microbenchmarks.

Table 2. Description of PAPI events under study.

PAPI name	PAPI number	Description
PAPI_TLB_TL	0x80000016	Total TLB misses
PAPI_TLB_DM	0x80000014	Data TLB misses
PAPI_L1_DCM	0x80000000	L1 data cache misses
PAPI_L2_DCM	0x80000002	L2 data cache misses
PAPI_CA_ITV	0x8000000d	Requests for a cache line intervention
PAPI_CA_INV	0x8000000c	Requests for a cache line invalidation
PAPI_CA_SHR	0x8000000a	Requests for exclusive access to a shared cache line
PAPI_CA_CLN	0x8000000b	Requests for exclusive access to a clean cache line

These events were chosen because of their association with the performance of the memory hierarchy, which usually defines the critical path in the overall performance of

an application. Furthermore, these events are primarily related to data references (as opposed to instruction references). The focus on data references was chosen because, in general, they exhibit poorer locality than instruction references (especially on high-performance computing applications where optimizing compilers play an essential role in pipelining and scheduling instructions). Therefore, analysis of these events usually has a higher payoff in terms of performance than analysis of those related to instruction references. In the case where an event is applicable to both data and instruction references, as in the case of the total TLB miss event described in Section 4.1, the event is studied with respect to data references only (instruction references are kept to a minimum).

Another aspect of the memory hierarchy addressed by this set of events is the *cache coherency* problem. As defined by **[13]**, the cache coherency problem arises when there are multiple writing units that have access to data that is replicated across multiple levels in the memory hierarchy. For example, in a uniprocessor, an I/O device can be reading main memory while the processor is writing the L1 data cache. If the cache is a write-back cache, then this situation can lead to data inconsistency and the I/O device reading stale data. In the case of a shared-memory multiprocessor, application performance may depend on the cache-coherency protocol, i.e., the protocol used to maintain the consistency of shared data (data shared by multiple processors). By far the most popular cache coherency protocols are of the type *write-invalidate*, which are implemented by associating a cache state with each cache line. The protocol ensures that a cache line is resident in only one cache, the cache of the owner processor, when it is being written, and

that a cache line can be resident in the caches of multiple processors only when it is being read. In the simplest protocol, the states are uncached, shared, and exclusive. The protocol defines the process that must be followed to change state and, thus, to ensure that writes are processed sequentially and that reads observe the most up-to-date version of a word. A line is in the *uncached* state when it is not contained in the cache of any processor. The *shared* state refers to a line that is included in the cache of more than one processor. While in this state, a cache line is read-only, i.e., it cannot be modified. Lastly, the *exclusive* state refers to a cache line that is resident only in the cache of a single processor; while in this state, the cache line can be read and written only by this processor, the owner. Note that a transition to the exclusive state may requires invalidation of cache-resident copies of the requested cache line; this is why the protocol is referred to as a write-invalidate protocol. The last four events of Table 2 are related to cache coherency protocols. These events are studied, in turn, in Sections 4.4 – 4.7, where relevant details of cache coherency protocols are discussed.

Probably the greatest challenge in validating events related to data memory references is dealing with *prefetching*. Nowadays, modern platforms use prefetching techniques to hide access latencies across the memory hierarchy in order to improve performance. Prefetching mechanisms provide the logic necessary to fetch data and/or instructions from the lower levels of memory before they are actually used so that they can be readily available by the time they are accessed by the processor. For example, **[14]** discusses a prefetching technique called *stream buffers*. In its simplest form, the technique consists of the prefetching of consecutive lines of data starting at a cache miss

address. The prefetched data is placed in a buffer. As it is needed, it is fed into the cache in FIFO order. Placing the prefetched data in the buffer instead of in the cache avoids polluting the cache with data that may never be needed. Accesses to the cache concurrently access the stream buffer as well. If a data reference misses in the cache but hits in the buffer, the cache can be reloaded in a single cycle from the stream buffer. Prefetching techniques are most beneficial when memory references exhibit a well-defined pattern of access, e.g., a sequence of accesses at a constant stride. Furthermore, in the case of the cache miss event, some performance counter implementations may not trigger the event if a cache miss is satisfied by a prefetching mechanism **[6]**. Prefetching techniques are usually employed in the memory hierarchy levels closest to the processor (i.e., L1 and L2 caches) because this is where they provide the greatest impact.

As described in Sections 4.2and 4.3, due to complexities associated with cache coherency, shared data is not prefetched. This realization permits a straight-forward validation of the L1 and L2 data cache miss events, which on a uniprocessor was not possible. Although prefetching does not appear to be commonplace w.r.t. TLBs, the possibility is not disregarded, and in Section 4.1 prefetching is considered in the discussion of the results for the data TLB miss event.

4.1 Data TLB miss event

In general, the data translation lookaside buffer (DTLB) miss event indicates that a virtual page that stores data was referenced and that its virtual-to-physical page mapping, i.e., its mapping to a corresponding physical page (or frame), is not resident in the DTLB. In general, this happens when a virtual page is accessed for the first time (a compulsory

miss) or when the TLB entry for a previously-referenced page has been replaced by that of another page (a capacity and/or conflict miss). Some platforms have separate TLBs for data and instructions and others have a unified TLB, which stores translations for both pages containing instructions and pages containing data. The events associated with TLBs vary among platforms. In the case of the platforms used in this study (refer to Chapter 3 for descriptions of these platforms), platforms A and B, which are based on the R10000 and the Power3 processors support only a total TLB miss event, i.e., given that they have a unified TLB (or at least it is suspected so based on the platform documentation), only a unified count is implemented. Platform C, based on the Itanium processor, which has an ITLB and two levels of DTLBs, supports separate events for DTLB misses and ITLB misses. It is not clear from the documentation if the DTLB miss event count accounts for only L1 DTLB misses or both L1 and L2 DTLB misses. Note there are no other PAPI-supported TLB-related events for this platform.

4.1.1 Validation microbenchmark

The validation microbenchmark for DTLB misses consists of code that executes one traversal of an array, accessing the array at regular page-sized strides. The array resides in multiple pages and the traversal generates a predictable number of compulsory DTLB misses. By focusing on compulsory misses only, the intrinsic features of the TLB (e.g., replacement policy and associativity) do not affect the resulting count. Furthermore, the portability of the benchmark is improved by not considering the platform-specific features of the TLB. Figure 3 shows the basic algorithm of the benchmark.

```
stride = PAGE_SIZE;
for (i=START_INDEX; i<MAX; i+=stride) {
    zz[i] = X_VALUE;
}
```

Figure 3. DTLB miss benchmark algorithm.

Three key pieces of information are needed to predict the DTLB miss count generated:

1. size of the pages that store the array,
2. identification of the location of the first byte of a page frame with respect to the beginning of the array data structure, and
3. the size of the array.

The page size and the stride with which the array is accessed determine the rate at which the benchmark generates DTLB misses. The identification of the first byte of a page frame is needed to perfect the prediction of total DTLB misses generated. The START_INDEX value in the algorithm above determines whether or not the first array reference maps to a previously unreferenced page. Setting START_INDEX so that the first array reference does map to the next unreferenced page causes references made at a stride equal to the page size to generate *n* compulsory DTLB misses, where *n* is the number of pages touched. Lastly, increasing the MAX value in the algorithm increases the number of pages of the array touched, thus increasing the expected count. The expected count is given by the number of executed for-loop iterations. The number of

instructions in the benchmark is small and does not significantly affect the event count associated with unified TLBs.

4.1.2 Acquiring parameters for the validation microbenchmark

As mentioned above, three pieces of information are needed to predict DTLB misses for the validation benchmark: page size, starting index, and array size. While the latter can be controlled by the user, the other two parameters may not be trivial to ascertain. Most modern platforms concurrently support multiple page sizes. For example, the Itanium documentation indicates support for the following page sizes: 4KB, 8KB, 16KB, 64KB, 256KB, 1MB, 4MB, 16MB, 64MB and 256MB **[6]**. However, documentation as to how a page size is selected is much more obscure and may even be OS-dependent. Furthermore, a starting index that guarantees a reference to the first word in the next unreferenced page is dependent upon the page size and compiler used (i.e., the prologue and epilogue sections of the executable file may include additional variables and constants that could offset the page to which the array maps).

To identify the page size used to store the array, several techniques are used. The first is a configuration microbenchmark based on Saavedra's methodology **[15]**. This microbenchmark accesses an array using a stride that is increased between iterations and records average read/write reference times. Several memory hierarchy characteristics, including data cache line size and associativity, page size, DTLB associativity and reachability, as well as corresponding latencies, can be deduced from graphing the average reference times. However, because of the increased efficiency with which some of the newer platforms hide memory hierarchy latencies, the results obtained are not

always easily understood. Figure 4 shows the results obtained for the MIPS R10K and the Itanium platforms.

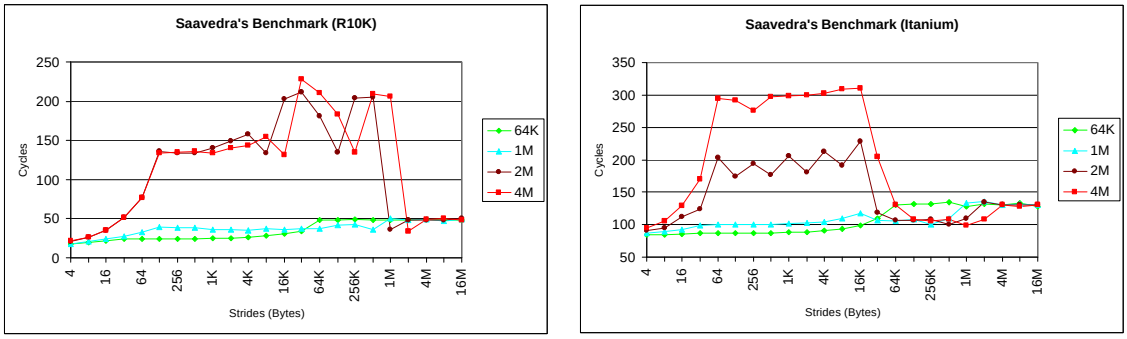


Figure 4. Saavedra's benchmark results for the R10000 and the Itanium platforms.

According to Saavedra's methodology, the page size can be deduced from these graphs by noticing the following. Let *b* be the size of the page and *s* the size of the stride at which the array is being referenced. If the array is large enough to fill up the DTLB and *s* is chosen so that it is smaller than the page size, then there will be *b*/*s* consecutive accesses to the same page and only the first one will generate a miss. The average reference time will reach its maximum when the stride is equal to the page size, i.e., *b*/*s*=1. An analogous argument is used to figure out the line sizes of the different levels of cache, with some overlapping effects happening in some regions. Isolation for a particular element of the memory hierarchy should be straightforward, given the difference in latencies. For the MIPS R10000 (R10K) platform, Figure 4 shows that the average reference time has its highest point at a stride of 32KB, which indicates a believable value for a page size. However, on the Itanium platform, this point occurs at a

stride of 64 bytes, which seems more like a cache line size. TLB behavior does not seem to be reflected in this graph.

The second technique used to identify the page size is a configuration microbenchmark, which also identifies the starting index to be used to cause the first array reference to map to the next unreferenced page. The benchmark is based on the assumption that a process' address space has a structure similar to that shown in Figure 5. The exact format of the address space depends on the specific platform, with some platforms including additional segments (e.g., header segments and shared-memory segments). The benchmark focuses on the data segment. It is assumed that constants/variables used in the benchmark are allocated space in the data segment in the following order: constant values at the beginning, followed by initialized variables, and dynamically allocated variables towards the end of the segment to facilitate growth. Furthermore, it is assumed that the order of variable declarations in the program and size of variables has an impact on the storage locations of the variables in the data segment.

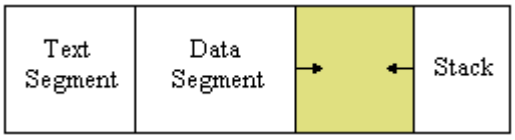


Figure 5. Storage of a process in memory.

The first objective of the benchmark is to force the array data structure, which is the focus of the validation benchmark, to be allocated last in the data segment. Figure 6 illustrates this idea. The second objective is to identify the smallest array index that will map to a frame that is entirely occupied by the array (i.e, the array index that maps to the

beginning of Frame 2 in Figure 6). The first objective is achieved by having the array as the only dynamically allocated variable in the benchmark and by declaring it last in the program sequence. Furthermore, since the array will be by far the largest data structure in the benchmark (i.e., it needs to occupy multiple pages in order to be useful in validating DTLB miss events), the assumption of placement according to variable size is also favored. Thus, the compiler should allocate space for the array towards the end of the data segment.

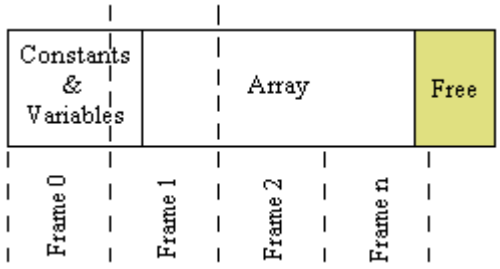


Figure 6. Distribution of variables in the data segment.

Initially the benchmark uses a hypothetical page size (based on Saavedra's results and/or platform documentation) to traverse the array, referencing the first and last array elements of every other page-sized region. The page-sized regions are defined by an offset or "padding" from the starting address of the array. Via PAPI, the number of generated DTLB misses is recorded. Successive variations of the benchmark use larger offsets and different page-size values. Figure 7 shows the basic algorithm for this benchmark, called the "padding" benchmark.

```
stride = PAGE_SIZE * 2;
for (i=START_INDEX; i<MAX; i+=stride) {
    zz[i] = X_VALUE;     // 1st element on a page range
    zz[i+PAGE_SIZE-1] = X_VALUE;     // Last element
}
```

Figure 7. Padding benchmark algorithm.

The DTLB miss counts reported for each variation of the benchmark identify the page-size and offset that forces aligned references in the validation benchmark, i.e., the page size and offset that produces a single DTLB miss for each pair of references. If pairs of references are not aligned, each pair will necessarily map to consecutive page frames and produce two misses. Figure 8 shows the results, for three different platforms, of executing the "padding benchmark" for 100 pairs of references that traverse the array. On the MIPS R10K, when using a 32KB page size, the recorded TLB miss count for all offsets except 28,012 bytes is approximately 200; for an offset of 28,012 bytes, the count drops to approximately half this amount, identifying START_INDEX for the validation benchmark. On the Power3, when using a 4KB page size, the recorded TLB miss count is 200 for all offsets except 2,260 bytes, at which point it drops by half this amount. Finally, on the Itanium, when using a 16KB page size, the recorded TLB miss count for all offsets except 16,364 bytes is approximately 200, dropping to half this amount for the offset of 16,364.

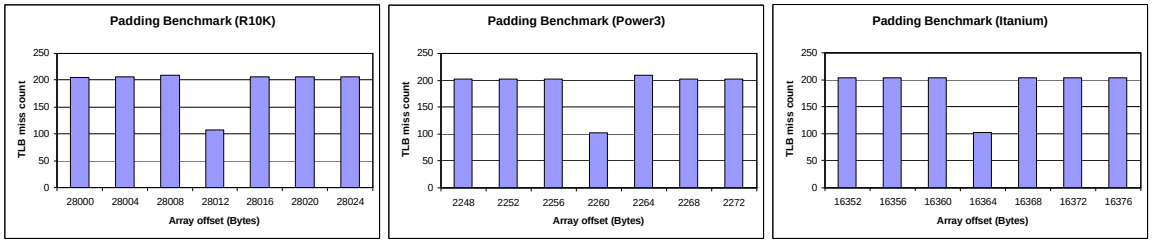


Figure 8. Padding benchmark results for 100 iterations.

4.1.3 Data collection

As mentioned in Section 2.2, the data collection process includes running 100 instances of each test case in a test suite and obtaining the average test-case count for each test case. The test suite used for the DTLB miss event includes test cases that are expected to generate from 1 to 10,000 DTLB misses, increasing in powers of 10. Attempts to include larger test cases (i.e., ones that produce more than 10,000 compulsory DTLB misses) resulted in segmentation faults on the platforms under study. This could be because either the array size exceeds the size of the defined virtual space of a process or some other system limitation was exceeded, (e.g., allocated user space).

4.1.4 Predicted vs. hardware-reported event counts

Validation of DTLB misses was performed for the experimental platforms A, B, and C, i.e., platforms (described in Chapter 3) based on the MIPS R10K, IBM Power3, and Intel Itanium processors. As mentioned before, the R10K and Power3 only support an event that counts both instruction and data TLB misses, while the Itanium supports events that count instruction and data TLB misses. Figure 9 presents a comparison of the predicted and hardware-reported DTLB miss event counts for the three platforms. In this graph, the y-axis represents the percentage difference between the reported and predicted

counts, and the x-axis represents the corresponding test case. Refer to Appendix B for the hardware-reported data.

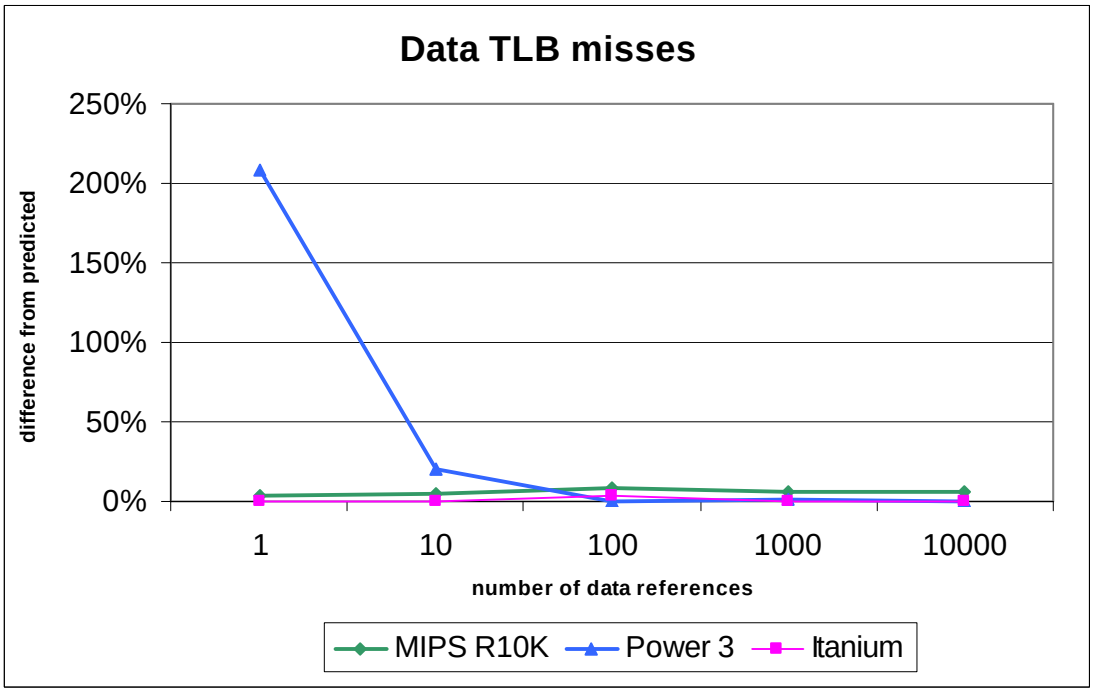


Figure 9. DTLB miss validation benchmark results.

For the MIPS R10K platform (platform A) the differences between predicted and reported counts range between 3% and 9%, the predicted being lower than the reported. At least part of the difference is likely attributable to overhead introduced by PAPI and to the fact that the monitored event reports a unified count of instruction and data TLB misses, although ITLB misses for the monitored section of code should be insignificant.

For the Power3 platform (platform B) the differences between predicted and reported counts start high for test cases with small numbers of data references, but tend to 0% as the number of data references increase. The hardware reports a consistent count of three

for the test case of one data reference, which results in the large percentage difference. The larger difference for tests cases with small numbers of references may be attributable in part to the fact that the Power3, like the MIPS R10K, supports only an event that counts both instruction and data TLB misses.

The Itanium platform (platform C) reported the best results. The worst test case exhibits a percentage difference of 3%. However, the hardware-reported data tends to reveal a consistent difference of 0% from the predicted, even for small numbers of data references. The stability of the results reported by this platform may be explained by the fact that this particular platform supports a DTLB miss count (i.e., it is not a unified count, as in the case of the other two platforms) and as a result, ITLB miss counts are not a factor.

Finally, another observation that can be made from these findings is that data prefetching mechanisms are not employed on any of these platforms at the DTLB level. This is because 1) the reported counts tend to be larger than the predicted; successful prefetching would have reduced the hardware-reported miss count, i.e., would have masked some of the predicted TLB misses, and 2) the percentage difference remains constant across the test suite or tends to zero; prefetching would have caused the percentage difference to continuously increase as test cases accessed more pages and prefetching masked more expected TLB misses.

4.2 L1 data cache miss event

In general, the L1 data cache (Dcache) miss event indicates that a cache line containing data (as opposed to instructions) has been requested by the processor and is

not resident in the cache. This happens when a previously unreferenced data cache line is accessed for the first time (a compulsory miss) or when a previously referenced line has been replaced by another (a capacity and/or conflict miss). Cache misses also can occur as a result of cache invalidations generated to maintain cache coherency in a shared-memory multiprocessor system (refer to Sections 4.4, 4.5, 4.6and 4.7 for related events).

Modern pipelined architectures execute multiple instructions concurrently. To facilitate the overlapping of instruction and data fetches, most of these platforms support separate L1 caches for instructions and data. Such is the case for the target platforms, in particular platform A, the MIPS R10K (refer to Section 3.1 for a detailed description). Furthermore, most platforms, including the R10K, also support the monitoring of a separate event for each of its L1 cache misses. The definition of this event, as described in the documentation of the R10K, is the following:

Primary data cache misses:

This counter is incremented one cycle after a request to refill a line of the primary data cache is entered into the secondary cache transaction processing logic. **[8]**

This definition seems to indicate that the L1 Dcache miss event is not triggered until the miss generates a request to the L2 cache.

4.2.1 Validation microbenchmark

Several attempts were made to predict the L1 Dcache miss event count. All of them are based on the validation microbenchmark presented in its most basic form in Figure 10. When executed, it traverses the array zz, which is stored in multiple memory blocks. The array is accessed at regular strides, where the stride is a multiple of the size of an L1

Dcache line. The goal is to produce a compulsory L1 Dcache miss per array access and to produce very few other L1 Dcache misses. Achieving this goal would allow a reasonably accurate prediction of the number of L1 Dcache miss events generated during the execution of the benchmark.

```
stride = L1_LINE_SIZE;
for (i=START_INDEX; i<MAX; i+=stride) {
    temp += zz[i];
}
```

Figure 10. L1 Dcache miss benchmark algorithm.

The basic L1 Dcache microbenchmark was unsuccessful in reaching this goal due to lack of knowledge of extremely efficient prefetching mechanisms implemented in modern microprocessors **[6]**. These prefetch mechanisms hide memory access latencies, especially with respect to memory-hierarchy levels closer to the processor, e.g., at the L1 and L2 cache levels. Execution of this microbenchmark on microprocessors with stream buffers **[14]** or other prefetching mechanisms resulted in hardware-reported miss counts that were significantly lower than predicted counts, approaching zero as the number of array references increased **[3]**. Since information about the prefetching mechanisms used by processor designers was not readily available, another strategy was employed, i.e., try to foil the prefetch mechanism **[6]**. A microbenchmark was designed that randomly accesses the array. In this case, to predict a count, the memory-access trace was captured and fed into a cache simulator configured without prefetching and with an LRU cache

replacement policy. According to the cross-platform results reported, hardware-reported counts fall within 10% of predicted counts once the array size supercedes the cache size.

This thesis presents a different attempt at foiling the prefetch mechanism. In this case, an event count can be predicted without cache simulations and, thus, can be more accurate since a simulator may not capture all of the details of the hardware. The related L1 Dcache microbenchmark, shown in Figure 11, foils prefetching through the use of shared memory accessed by two processors concurrently executing the benchmark in a multiprocessor. It was hypothesized that prefetching mechanisms are not employed when using shared memory because of the need to maintain cache coherency among processors and because of the performance penalties associated with prefetching data that is being used by another processor; as discussed below, the hypothesis was proven to be correct. As with the benchmarks discussed above, since only compulsory misses are generated, no assumptions or knowledge about the cache replacement policy or configuration are required to make a prediction, and cache pollution due to other concurrently executing processes is not an issue.

The shared-memory benchmark is derived from the benchmark of Figure 10 by placing the array in shared memory and having two processes, each running on separate processors, traverse the array and generate the compulsory misses. The array is divided equally between the two processors. During the first phase of execution each process initializes one half of the array; during the second phase, the half initialized by one process is traversed by the other. For the R10K platform used, process execution on

separate processors was ensured by means of the *sysmp* system call, which is described in Table 1 of Section 3.1.

```
stride = L1_LINE_SIZE;
HALF_SIZE = ARRAY_SIZE / 2
// Phase 1
// Process 0 has OFFSET == 0, Process 1 has OFFSET == HALF_SIZE;
for (i = 0; i < HALF_SIZE; i += stride) {
    zz[i+OFFSET] = 0; // Initialize array
}
// Phase 2
// Process 0 has OFFSET == HALF_SIZE, Process 1 has OFFSET == 0;
for (i = 0; i< HALF_SIZE; i += stride) {
    temp += zz[i+OFFSET];
}
```

Figure 11. Shared-memory L1 Dcache miss benchmark algorithm.

The first phase of the benchmark accomplishes two things: 1) initializes (writes) the array so that the read instructions of the second phase are not removed during compile optimization and 2) guarantees that each half of the array is exclusively owned by the cache of the initializing processor. Note that only array elements that map into the stride need to be initialized. Array elements in between stride elements are not accessed. They are brought into the cache on cache misses generated by an access to a stride element.

Phase two of the benchmark causes each process to traverse, at regular strides, the half of the array initialized by the other processor. Each reference to the array during the second phase produces a compulsory cache miss. It should be noted that a barrier is needed between phases in order to ensure that both processes are synchronized when they start executing the second phase. If synchronization is not enforced between phases, a process could potentially start referencing elements that have not been initialized. The predicted event count is equal to the number of for-loop iterations executed in this phase.

4.2.2 Data Collection

As mentioned in Section 2.2, the data collection process includes running test cases for 100 runs and obtaining the average test-case count. The test suite used for the L1 Dcache miss event includes test cases that are expected to generate from 1 to 1,000,000 array references and L1 Dcache misses. The counts generated by both processors are monitored separately; that is, one test case is executed to monitor the behavior of processor 0 and a second execution of the same test case is executed to monitor the behavior of processor 1.

4.2.3 Predicted vs. hardware-reported event counts

Figure 12 presents the predicted and hardware-reported L1 Dcache miss event counts for both processors involved in the execution of the benchmark. The y-axis represents the percentage difference between the hardware-reported counts and the predicted counts, and the x-axis represents the number of array references generated by the benchmark. Refer to Appendix B for the hardware-reported data.

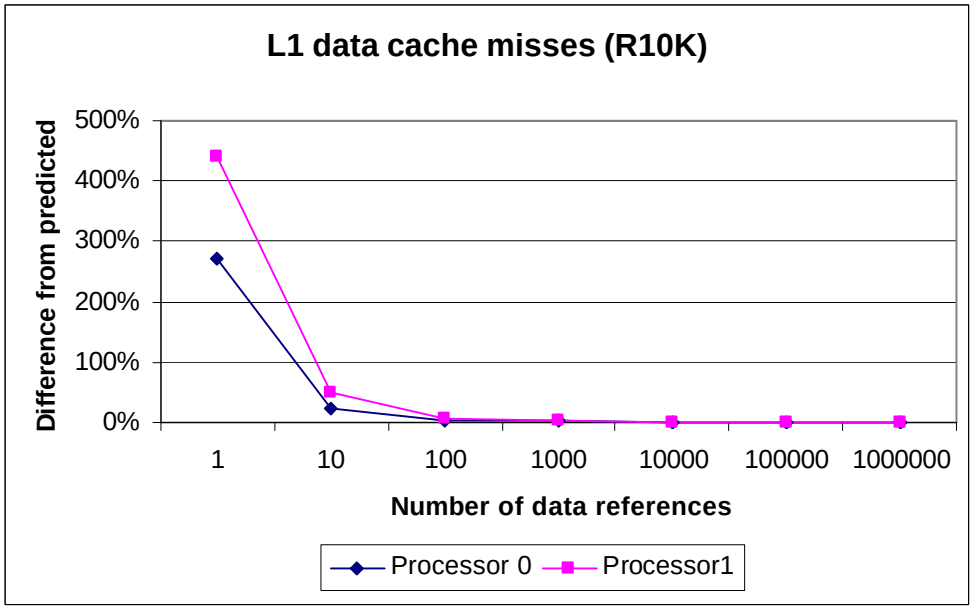


Figure 12. L1 Dcache miss validation benchmark results.

The results illustrate that the difference between predicted and reported counts is large for test cases with numbers of references below 10 but rapidly decreases and approaches zero as the number of references increases. A possible source for the discrepancy is the overhead introduced by PAPI, which is more notable in the smaller test cases. The benchmark is designed to generate compulsory misses due to array accesses. Thus, cache pollution due to other processes should not affect the count. Furthermore, PAPI segregates between the misses generated on behalf of the benchmark process and other processes executing concurrently. However, PAPI data (i.e., variables and constants set and used by the PAPI interface) referenced by the benchmark via PAPI start and stop calls and PAPI housekeeping routines (refer to Appendix A for description) can be susceptible to cache pollution from other processes and from the execution of the

benchmark itself. This may be the cause of some of the misses included in the hardware-reported counts. In addition, the fact that the standard deviation of the experiments remains fairly constant (below three for both processors for all test cases) supports the idea that the overhead introduced by PAPI is the predominant source for deviation, as opposed to other external factors such as kernel processes or other processes in the processor's workload. Refer to Appendix B for the hardware-reported data. The latter could cause additional cache misses but their affect would vary depending upon their level of perturbation for each run. Finally, a standard deviation value of below three may be good for test cases with a number of references of 100 and above, but may not be reliable enough for test cases with fewer references. In other words, the granularity of the monitored code executed when test cases are below 100 data references is not sufficient for PAPI to provide a reliable count.

Figure 12 also shows a difference between the hardware-reported event counts of the two processors. This may be due to the variations in the workload of each processor and the resultant perturbation caused by other processes. For example, if kernel processes generally run on processor 0 and shorter-running, user processes mainly run on processor 1, then the longer-running kernel processes (i.e., processes, such as daemons, that repeatedly execute in the background for longer periods) may cause a greater perturbation on processor 0. However, this type of perturbation should be insignificant, especially w.r.t. variations of the benchmark that generate large numbers of references. The results seem to concur with this hypothesis, given that the gap separating the graphs associated with the two processors approaches zero as the number of references increases.

4.3 L2 data cache miss event

The L2 Dcache miss event is similar to the L1 Dcache miss described in Section 4.2. However, in the case of the L2 cache, some platforms implement separate L2 caches for data and instructions and others implement a unified L2 cache, as is the case in the R10K platform (i.e., platform A described in Section 3.1), which is used to validate this event. Nonetheless, this platform does support separate events for L2 Icache misses and L2 Dcache misses. The definition of the L2 Dcache miss event is described in the documentation of the R10K as:

Secondary data cache misses:

This counter is incremented the cycle after the second quadword of a data cache line is written from the main memory, while the secondary cache refill continues. **[8]**

This definition indicates that the event is triggered while the data request that missed on the L2 cache is being satisfied by main memory. It also indicates that the cache is refilled by a quad-word at a time. Thus, eight transactions are needed to load the 128-byte line of the L2 cache.

The method used to predict and collect hardware-reported L2 Dcache miss event counts mirrors that used to for the L1 Dcache miss event. The L2 Dcache miss validation microbenchmark essentially is the same as its L1 Dcache counterpart (refer to Figure 11 in the previous section). The only difference is that the L1_LINE_SIZE constant is replaced by the L2_LINE_SIZE constant. The test suite includes test cases that generate an expected count of L2 Dcache misses that range from 1 to 1,000,000.

4.3.1 Predicted vs. hardware-reported event counts

Predicted and hardware-reported counts are presented in Figure 13, where the y-axis represents the percentage difference between the reported and predicted counts and the x-axis represents the test cases, i.e., the number of data references or L2 Dcache misses generated by the test cases. Refer to Appendix B for the hardware-reported data.

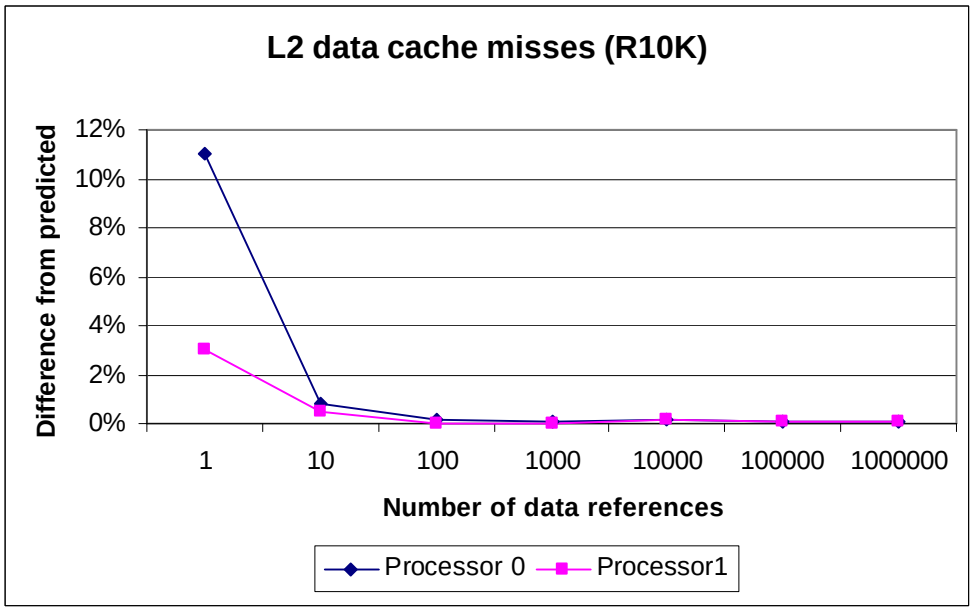


Figure 13. L2 Dcache miss validation benchmark results.

These results are similar to the results obtained for the L1 Dcache miss event. The discrepancy between predicted and hardware-reported counts is greater for smaller test cases and approaches zero as the number of data references increases. There is an important difference, however; the maximum discrepancy for small test cases is several orders of magnitude smaller than it is for the L1 Dcache miss event. This behavior is expected—it is due to the difference in the reachability of the different size cache lines.

In other words, assuming that PAPI data exhibits good locality of reference, the larger L2 cache line size (128 bytes) should make the external perturbations shown in the previous figure less significant than they are for the L1 Dcache (with a line size of 32 bytes).

4.4 Cache intervention request event

On a multiprocessor system, the cache intervention event is related to the cache coherency family of events. The cache intervention event in a directory-based system occurs in the following scenario. Suppose a cache line is held exclusively at some processor's cache -- this processor is considered the exclusive owner of the line. In this case, if the cache has a write-back policy and the line is dirty (i.e., it was modified while cache resident), then the main memory copy of the line is stale. When another processor generates a cache miss for the same line, the cache coherence unit ensures that the processor gets the up-to-date data by requesting a write-back from the owner processor's cache, which subsequently is sent to the requesting processor. This is called an *intervention request* to the owner cache **[13]**. A large number of intervention requests indicates that processors frequently are reading the same data (or data that coincidently maps to the same cache line) that other processors are modifying. Identifying such a producer-consumer situation and improving its performance by introducing synchronization of reads and writes or distributing the common data across multiple cache lines can reduce the number of intervention requests. Such corrective actions can alleviate the writing processors from having to reinstate exclusive access before rewriting the data and allowing the reading processors to retain read elements longer in their caches.

The platform used to validate this event is platform A, the R10K platform described in Section 3.1, which implements a directory-based cache coherency protocol at the L2 cache level. The platform documentation describes the cache intervention request event as follows:

External intervention requests:

This counter is incremented on the cycle after an external intervention request enters the secondary cache transaction processing logic. **[8]**

This indicates that the event is triggered as soon as the intervention request is received by the logic of the targeted cache and not after it has been satisfied. It should be noted that even though this work is focusing on data references, since the R10K L2 cache is unified, this event potentially can be triggered by both data and instruction references. However, since instruction cache lines are hardly ever modified (i.e., self-modifying code is not common), the probability of instruction references triggering this event is near zero.

4.4.1 Validation microbenchmark

In order to validate the count for the cache intervention request event, a microbenchmark that involves two concurrently executing processes is employed. Each process runs on different processors, processors 0 and 1, and both processes traverse a shared array. During the first phase of the benchmark, processor 0 initializes the array and, as a result, its cache exclusively owns the array. During the second phase, processor 1 reads the array, generating cache intervention requests to the cache of processor 0, which is the monitored processor. The basic benchmark follows in Figure 14.

```
stride = LINE_SIZE;
// Phase 1
// Process 0 initializes array, exclusively owned in its cache, Process 1 sets counter
if (myid == 0) {
    for (i = 0; i < ARRAY_SIZE; i += stride)
        zz[i] = 0;
}
else
    counter = 0;
// Phase 2
// Processor 1 traverses initialized array, causing interventions on Processor 0.
// Processor 0 busy-waits until Processor 1 finishes traversal.
if (myid == 0) {
    while (counter < NUM_ITER);
}
else {
    for (i = 0; i < ARRAY_SIZE; i += stride) {
        zz[i] = 0;
    counter++;
}
```

Figure 14. Cache intervention requests benchmark algorithm.

The array traversal by both processes is made with a stride of a cache line size. Additional references to the same line will not cause additional interventions. Notice that the shared variable, *counter*, which causes processor 0 to busy-wait while processor 1 generates intervention requests to processor 0, is accessed by both processors, but is modified only by processor 1. Since only processor 0 is monitored for cache intervention requests, the intervention requests for processor 1 generated by processor 0 referencing the *counter* variable are not included in the hardware-reported count. Only the intervention requests for processor 0 generated by processor 1 writing the zz array are included in the reported count. Thus, the predicted event count is NUM_ITER, the number of elements of the zz array written by processor 1. However, the prediction will hold only as long as the array size does not surpass the cache size. If the array is larger than the cache size, the first cache lines accessed during phase one will be replaced by ones accessed later; by the time the execution of the second phase starts, only the elements that remain in the cache of the initializing processor will trigger the event.

It also should be noted that a barrier is needed between phases in order to ensure that processor 1 does not start traversing the array before it has been completely initialized by processor 0; otherwise, the expected count could diverge from the predicted due to unsynchronized process execution.

4.4.2 Data collection

As mentioned in Section 2.2, the data collection process includes running test cases for 100 runs and obtaining the average test-case count. The test suite used for the cache intervention request event includes test cases that are expected to generate from 1 to

10,000 events of this kind, increasing by powers of 10. Including larger test cases is not productive because the array size must be smaller than the cache size. The R10K's L2 cache stores a total of 8192 cache lines. Two test cases greater than 8192 are included in the test suite to reaffirm the microbenchmark's limitation on test case size.

4.4.3 Predicted vs. hardware-reported event counts

Figure 15 presents the cache line intervention event counts predicted and the event counts reported by the hardware; the y-axis represents the percentage difference between the reported count and the predicted count, and the x-axis represents the number of array references. Refer to Appendix B for the hardware-reported data.

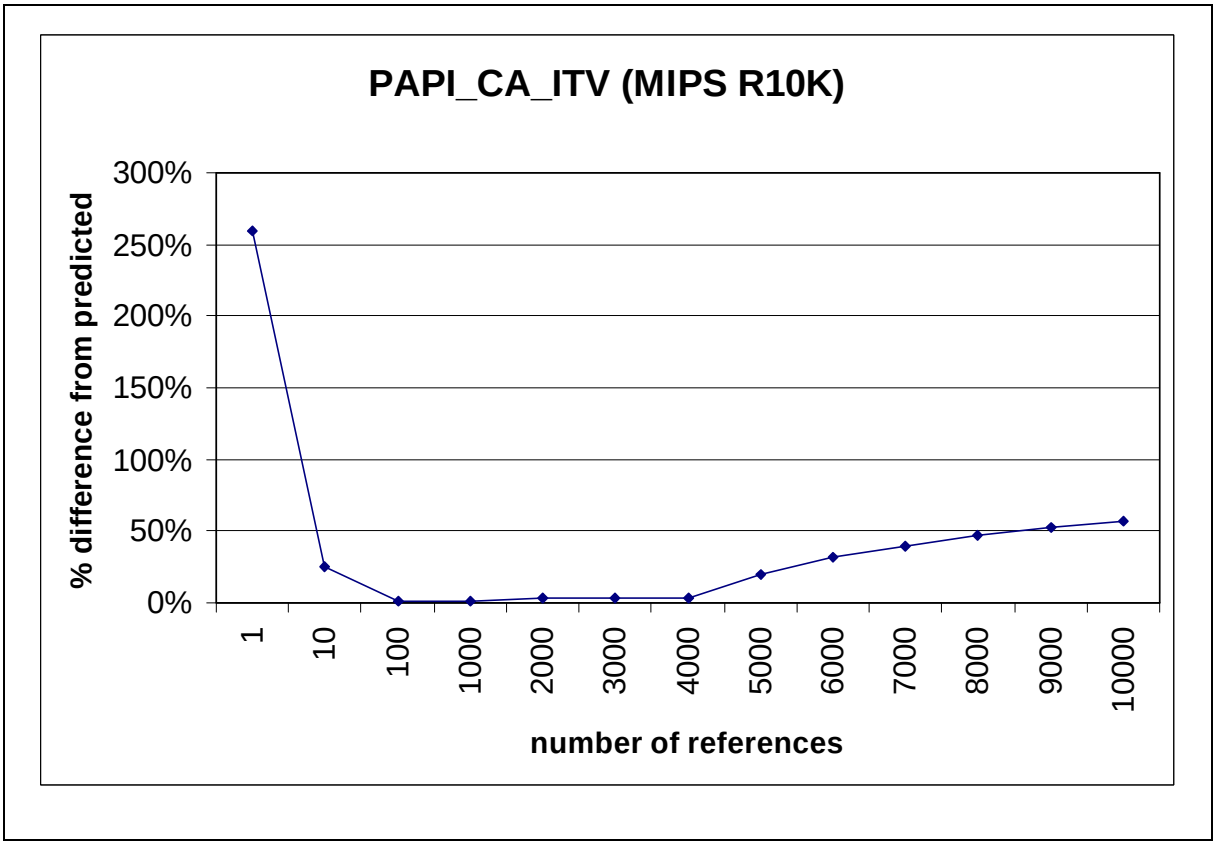


Figure 15. Cache intervention requests validation benchmark results.

The results indicate that a small number of reported events are not included in the predicted counts. This small difference is more noticeable in small test cases, which reference few data cache lines. This could be due to cache pollution from other processes, which may replace array cache lines in the initializing processor before the other processor reads them. The difference starts to widen at about 4000 data references. Cache consistency is kept at the L2 cache level and the L2 cache size on the R10K is 1MB with a line size of 128 bytes. This means that the total number of L2 cache lines is 8192; furthermore, the L2 cache is a unified cache, which means that some lines may store instructions. While it is not clear exactly how many lines hold data, the results indicate that the prediction holds until the cache is half full of data; after that point the prediction starts to decrease in accuracy. The platform documentation does not mention how cache lines are distributed among instructions and data at the unified L2 cache. Even assuming the cache lines are assigned in a first come first served basis, the number of cache lines assigned to data would still be difficult to ascertain due to external processes executing concurrently.

4.5 Cache invalidation request event

On a multiprocessor system, the cache invalidation event is related to the cache coherency family of events. By far, the most popular cache coherency protocol type is the write-invalidate. As described by **[13]**, this method implements cache coherency by giving a processor exclusive access to a data item before writing to it. Exclusive access of a data item is achieved by defining its cache state as *exclusive* and invalidating copies resident in other processors' caches on a write. The invalidation of these other copies

forces other reading processors or a writing processor to generate a cache miss and fetch a fresh copy when next accessing the cache line. In this case, a reading processor generates a read miss, which results in fetching the copy updated by the writing processor and changing the line's state in both processors' caches to *shared*. On the other hand, if the line is held in the *exclusive* state, then the processor generating the write miss (i.e., the "new" owner) causes the updated copy to be fetched from the "previous" owner and the line to be invalidate in the previous owner's cache. Accordingly, write serialization is also enforced by the protocol.

A cache line invalidation request occurs when a processor is signaled to invalidate one of its cache lines. This event can be helpful in identifying performance bottlenecks in multiprocessor applications due to ping-ponging (i.e., processors alternately invalidating each other's cache lines). In this case performance can be improved by introducing synchronization of writes and distributing the common data across multiple cache lines. Such corrective actions can alleviate the writing processors from having to reinstate exclusive access before rewriting the data.

The platform used to validate this event is the R10K platform described in Section 3.1, which implements a directory-based cache coherency protocol at the L2 cache level. The platform documentation describes the cache invalidation request event as follows:

External invalidation requests:

This counter is incremented on the cycle after an external invalidate request enters the secondary cache transaction processing logic. **[8]**

This indicates that the event is triggered as soon as the invalidation request is received by the logic of the targeted cache and not after it has been satisfied. It should be noted that even though this work is focusing on data references, since the R10K L2 cache is unified, this event can potentially be triggered by both data and instruction references. However, since instruction cache lines are hardly ever modified (i.e., self-modifying code is not common), the probability of instruction references triggering this event is near zero.

4.5.1 Validation microbenchmark

The validation microbenchmark for this event involves two processes, each executing on separate processors. It forces the occurrence of ping-ponging between them. One processor, say processor 0, writes to a shared variable and requests exclusive access to the appropriate cache line. Next, processor 1 writes to the same shared variable and causes the invalidation of the line in the cache of processor 0. This sequence of events is repeated a determined number of times to generate a predicted count. The key factor in generating the intended event is the synchronization between the processors. Processor 0 has to wait until its cache line is invalidated before attempting to write and invalidate the other processor's cache line. If write serialization is not enforced in the benchmark, then the cache coherency protocol will enforce it, but the non-determinism resulting from this will complicate the analysis of the benchmark. Enforcing synchronization by means of barrier implementations available on the target platform (e.g., system calls or library calls) proved not to be ideal because it appears that the barrier causes an unpredictable number of invalidations or causes a process to block, which may affect the event count. A

predictable number of invalidations is generated by implementing the barriers in the benchmark itself. This was done by having one processor, say processor 0, busy wait until the other processor, say processor 1, writes the shared variable. Subsequently, processor 1 busy waits until the processor 0 writes. Busy waiting is implemented by having the processor repeatedly read the shared variable until the appropriate condition is satisfied, i.e., the other processor changes its value. Reading the shared variable does not cause invalidations in the writing processor's cache and, as a result, busy waiting does not affect the event count. The basic benchmark follows in Figure 16.

In the benchmark, each processor is assigned a constant value (e.g., processor 0 gets constant X and processor 1 gets constant Y) with which it sets the value of the shared variable and busy waits until the variable is modified. Execution of the writes and busy-waits alternates between the processors. In order to enforce determinism in the alternation sequence, the shared variable should be initialized before either of the processes enters its for-loop. The initializing value should be the constant associated with the processor that is intended to execute the busy-wait instruction first. Initializing the shared variable to any other value potentially could create a race condition between the processes at the write instructions. This would have to be handled by the hardware and could introduce non-determinism with respect to generating the intended event. Only one processor is monitored at a time, and the predicted count is equal to the number of for-loop iterations executed; num_iter in the case of the benchmark of Figure 16.

```
// Shared variable initialized to constant Y
s = Y;
// Processor 0 invalidates Processor 1 cache line and busy waits
if (myid==0) {
    for (i=0; i<num_iter; i++) {
        s = X;
        while (s == X);
    }
// Processor 1 busy waits until processor 0 invalidates its line and then reciprocates
} else {
    for (i=0; i<num_iter; i++) {
        while (s == Y);
        s = Y;
    }
}
```

Figure 16. Cache invalidation requests benchmark algorithm.

4.5.2 Data collection

As mentioned in Section 2.2, the data collection process consists of running test cases for 100 runs and obtaining the average test-case count. The test suite used for the cache invalidation request event includes test cases that are expected to generate from 1 to 1,000,000 events of this kind.

4.5.3 Predicted vs. hardware-reported event counts

Figure 17 presents the cache line invalidation request event counts predicted and the event counts reported by the hardware. The y-axis represents the percentage difference between the reported counts and the predicted counts, and the x-axis represents the number of cache line invalidation requests generated by the validation benchmark. Refer to Appendix B for the hardware-reported data.

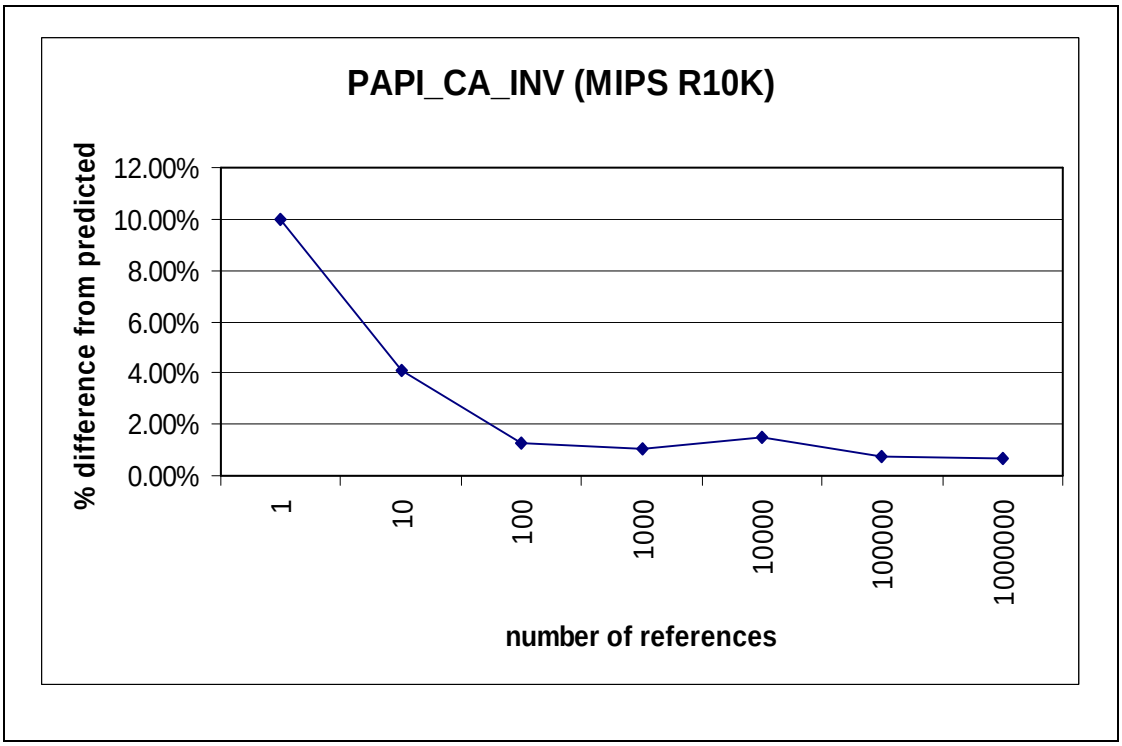


Figure 17. Cache invalidation requests validation benchmark results.

The results indicate that there is a small perturbation throughout the test cases that is more significant when the intended number of cache invalidation requests is small. In general, the percentage difference is positive, which indicates that the reported count is greater than or equal to the predicted count (refer to Section 2.3 for the percentage

difference formula used). The source of the additional invalidation requests could be due to write instructions executed speculatively and then discarded when a branch is mispredicted. There are two branch instructions in the benchmark: a conditional branch used to implement the for-loop and a conditional branch used to implement the busy-wait. Although the for-loop branch instruction is expected to be predicted correctly most of the time, especially for larger test cases, the same cannot be expected for the busy-wait branch. This is because the number of times that the busy-wait branch instruction is executed varies from iteration to iteration: it depends on the behavior of the other process. Consequently, the number of times that the busy-wait branch instruction is consecutively resolved in the "taken" direction varies as well. This behavior could result in inefficient performance on behalf of the branch prediction mechanisms employed. Furthermore, the cache coherence protocol has to ensure exclusive access to data before a write instruction commits (i.e., the invalidation request has to be processed before the actual commit of data to a physical register or memory). Thus, if the processor architecture allows speculatively executed invalidation requests, the following situation could arise. A speculatively executed write instruction generates an invalidation request so that it can modify the targeted data. After the invalidation request is processed, the write resides in the reorder buffer awaiting the commit of a dependent branch instruction that is yet to be resolved. If the pending branch was mispredicted, an "extra" invalidation request event, not associated with an actual write, occurs. If this is the case, the hardware-reported count could include speculatively-executed invalidation requests generated on behalf of discarded instructions.

Figure 18 shows the number of mispredicted branch instructions reported for the benchmark. The mispredicted branch event count was collected using PAPI, where both the request for cache invalidation and the mispredicted branch instruction events were monitored concurrently on each run. The x-axis represents the benchmark test case size and the y-axis represents the average of the reported event counts for branch mispredictions for 100 runs of each test case. The y-axis is in logarithmic scale to show the proportional increase w.r.t. test-case size. As the test-case size increases, so does the number of mispredicted branches. If the hypothesis of speculatively-executed invalidation requests is correct, as the number of mispredicted branches increases there is more potential for speculatively-executed invalidation requests to be generated.

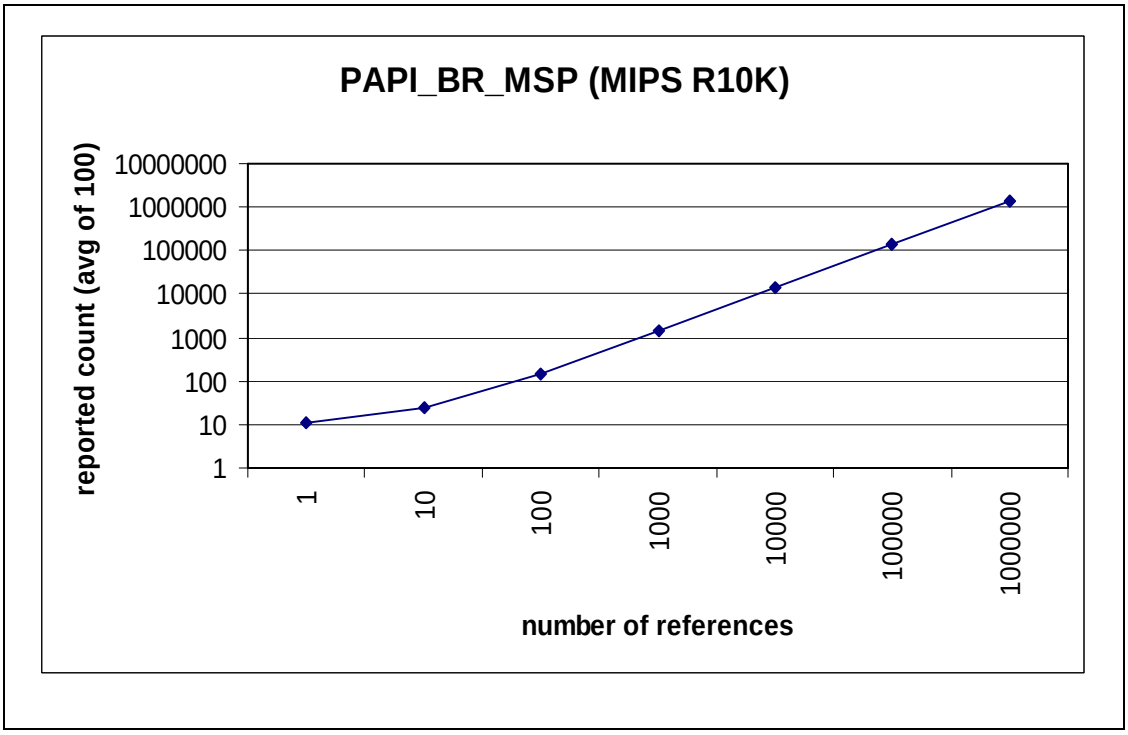


Figure 18. Branch mispredictions on the cache invalidation requests benchmark.

4.6 Request for exclusive access to a shared cache line

On a multiprocessor system, the request for exclusive access to a shared cache line event is part of the cache coherency protocol family of events. This event should be triggered when a processor attempts to write a cache line that is in the *shared* state. When this happens, the line is invalidated in the caches of all other processors in order to maintain cache coherency In the case of the R10K platform described in Section 3.1, the state of the cache line is set to *dirty exclusive* and is resident only in the cache of the writing processor, which is now considered the "owner" of the cache line. This event can be helpful in identifying performance bottlenecks in multiprocessor applications that are due to the ping-pong effect (i.e., processors alternatively invalidating each other's cache lines). If identified, this effect could be eliminated by synchronizing the reads and writes of executing threads/processes or, in the case of false sharing (i.e., the case where different processors access different variables that map to the same cache line), by offsetting data to force references to different cache lines.

This event is validated on the R10K platform, which implements a directory-based cache coherency protocol at the L2 cache level. The platform documentation describes the requests for exclusive access to a shared cache line as follows:

Stores or prefetches with store hints to shared secondary cache blocks:

This counter is incremented on the cycle after a request to change the shared state of the targeted secondary cache line to dirty exclusive is sent to the secondary cache transaction processing logic. **[8]**

This indicates that the event is triggered as soon as the request is sent from the originating processor and not after it has been received by the target processor.

4.6.1 Validation microbenchmark

The validation microbenchmark for the request for exclusive access to a shared cache line event, shown in Figure 19, is executed by two processes running on two different processors. During the first phase of the benchmark, one processor (e.g., processor 0) initializes elements of the array. During the second phase, the other processor (e.g., processor 1) consecutively reads and writes the same elements of the array. A barrier is needed between phases in order to ensure that processor 1 does not start traversing the array before it has been completely initialized by processor 0, otherwise the hardware-reported count could diverge from the predicted due to unsynchronized access.

The array is stored in multiple memory blocks allocated in shared memory. As a result, when processor 0 attempts to initialize an element of the array, it generates a request for exclusive access to the referenced cache line. Since the line is not resident in any other processor's cache, it does not generate the subject event. This causes the state of the accessed cache line, stored only in its cache, to become *dirty exclusive*. When processor 1 read misses on one of these cache lines, the state of the line, in both caches, becomes *shared*. Subsequently, when processor 1 attempts to write to a *shared* cache line, it generates a request for exclusive access to the referenced cache line but now the referenced cache line is in the *shared* state so the subject event is generated. These write misses cause the state of the associated lines to change to *dirty exclusive* and invalidate

the corresponding cache lines in all other processors. Cache coherency protocols typically work at the cache line level and not at the individual word level. Therefore, the array is accessed at strides equal to the cache line size. Only one word in a line needs to be written in order to generate the event; modifications to other words in the same line may not trigger the event. The goal is to produce one event per access.

```
stride = LINE_SIZE;
// Phase 1
// Process 0 initializes array, cache line becomes “dirty exclusive”
for (i = 0; i < ARRAY_SIZE; i += stride) {
    zz[i] = 0;
}
// Phase 2
// Processor 1 traverses the initialized array
for (i = 0; i< ARRAY_SIZE; i += stride) {
    temp += zz[i];  // Read shared array, cache line becomes shared
    zz[i] = 1;  // Write shared cache line, request exclusive access

}
```

Figure 19. Requests for exclusive access to a shared cache line benchmark algorithm.

The first phase of the benchmark accomplishes two tasks: 1) it initializes the array so that the read instructions of the second phase are not removed during compile

optimization and 2) it guarantees that the cache lines that store the array are exclusively owned by the cache of the initializing processor. As mentioned above, only array elements that map into the stride need to be initialized. Phase two of the benchmark causes the other processor to traverse the initialized array at regular cache-line strides.

The predicted count is equal to the number of for-loop iterations executed in the second phase of the benchmark; however, the prediction only holds as long as the array does not surpass the number of lines that can be simultaneously resident in the cache. If the array is larger than the cache, the first elements initialized in phase one will be replaced by later-referenced ones. By the time the execution of the second phase starts, only the elements that remain in the cache of the initializing processor can trigger the event.

4.6.2 Data Collection

As mentioned in Section 2.2, the data collection process consists of running test cases for 100 runs and obtaining the average test-case count. The test suite used for the requests for exclusive access to a shared cache line event includes test cases that are expected to generate from 1 to 10,000 events of this kind. Including larger test cases is not productive because of the benchmark requirement mentioned above (i.e., the array size must be smaller than the cache size). The target platform's (i.e., the R10K's) L2 cache stores a total of 8192 cache lines. Two test cases greater than 8192 are included in the test suite to reaffirm the microbenchmark's limitation on test case size.

4.6.3 Predicted vs. hardware-reported event counts

Figure 20 presents the predicted and hardware-reported counts. The y-axis represents the percentage difference between the reported counts and the predicted counts, and the x-axis represents the test cases, i.e., the number of intended requests for exclusive access to a shared cache line. Refer to Appendix B for the hardware-reported data.

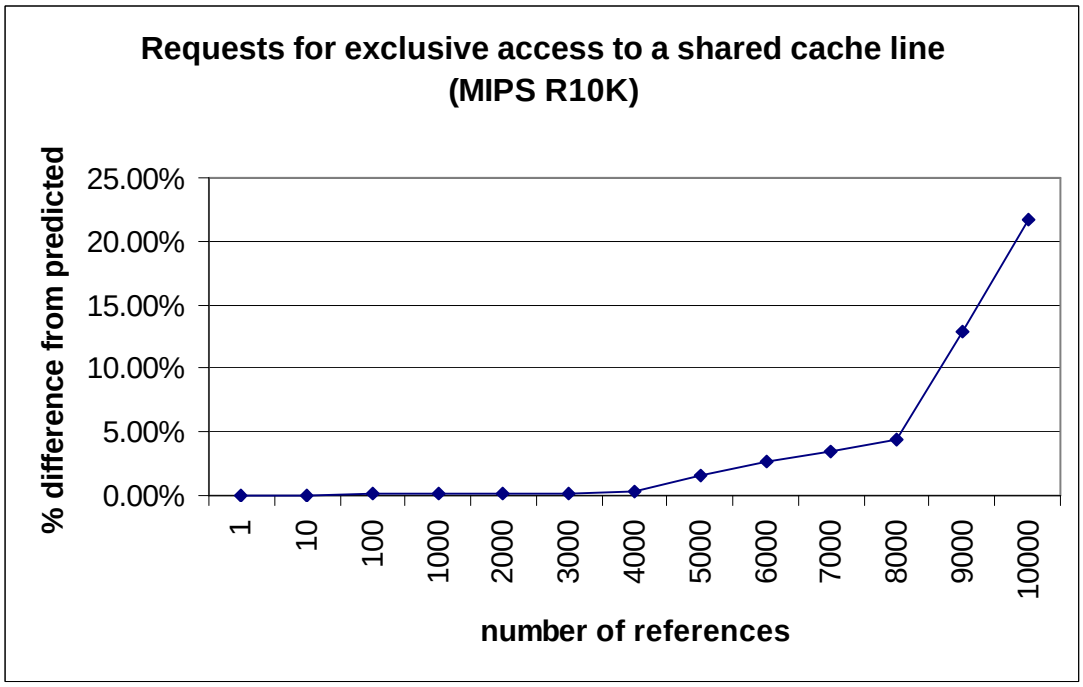


Figure 20. Requests for exclusive access to a shared cache line validation benchmark results.

The predicted and reported counts are equal for small test cases. The difference between these counts starts to widen at about 4000 data references and notably jumps for test cases that generate more than 8000 data references. As mentioned above, cache coherency is kept at the L2 cache level and the L2 cache size on this platform is 1MB with a line size of 128 bytes. This means that the total number of lines for the L2 cache is

8192. Furthermore, the L2 cache is a unified cache, which means that some lines can be used to store instructions. While it is not clear exactly how many lines are used for data, the results do seem to indicate that the prediction holds until half the cache contains lines associated with the referenced array. After this point, although the accuracy of the prediction starts to decrease, the difference between the predicted and reported counts holds below five percent until the test case of 8000 data references. The difference varies widely once the array exceeds the size of the cache. This is expected since the first cache lines initialized during phase one will be replaced by ones accessed later; by the time the execution of the second phase starts, only the elements that remain in the cache of the initializing processor will trigger the event.

4.7 Request for exclusive access to a clean cache line

On a multiprocessor system, the request for exclusive access to a clean cache line event is part of the cache consistency protocol family of events. On the R10K platform described in section 3.1, this event is triggered when a processor attempts to write a cache line that is in the *clean exclusive* state. A cache line is said to be in the *clean exclusive* state when the line is in the cache of only one processor, which is considered the "*owner*", and the line has not been modified during this cache residency. The *clean exclusive* state is not supported by some multiprocessor platforms. Instead it is considered to be part of the more general "*shared*" state, which is the state of a line that was cached as a result of a read miss and has not been modified during this cache residency. When the *clean exclusive* state is supported, it can help reduce the cache latency generated by read-modify-write operations and the cache coherency protocol. This is because the *clean*

exclusive state indicates that there is no need to invalidate any other processor's cache when the line is modified and, therefore, there is no need to verify the directory. If only the *shared* state is used, the directory has to be referenced every time a line is modified to verify the existence of replicated cache lines that need to be invalidated.

The platform used to validate this event is the R10K platform described in Section 3.1, which implements a directory-based cache coherency protocol at the L2 cache level. The platform documentation describes the requests for exclusive access to a clean cache line event as follows:

Stores or prefetches with store hint to clean exclusive secondary cache blocks:

This counter is incremented on the cycle after a request to change the clean exclusive state of the targeted secondary cache line to dirty exclusive is sent to the secondary cache transaction processing logic. **[8]**

The definition indicates that the event is triggered as soon as the request is sent from the originating processor and not after it has been received by the target processor.

4.7.1 Validation microbenchmark

The validation microbenchmark for the request for exclusive access to a clean cache line event consists of code that executes one traversal of an array that is stored in multiple memory blocks in shared memory. The traversing processor loads a block of the array into its cache so that it can read it, modify it, and then write it. This forces the cache line to change from the *clean exclusive* state to the *dirty exclusive* state and trigger the event under study. The array is accessed at regular strides of the size of a cache line. Only one word in the line needs to be manipulated in order to generate the intended count. Based

on the discussion presented for the validation of the L1 and L2 data cache miss events described in Sections 4.2 and 4.3, data in shared memory is not prefetched. The shared array is divided in half, and each process traverses one half. The basic benchmark is shown in Figure 21.

```
stride = LINE_SIZE;
HALF_SIZE = ARRAY_SIZE / 2
// Process 0 has OFFSET == 0;
// Process 1 has OFFSET == HALF_SIZE;
for (i = 0; i < HALF_SIZE; i += stride) {
    temp += zz[i+OFFSET] // Read uncached line: Line is clean exclusive
    zz[i+OFFSET] = 0; // Write line: Line is set to dirty exclusive
 }
```

Figure 21. Requests for exclusive access to a clean cache line benchmark algorithm.

Note that the array is not initialized by either process. If the array is explicitly initialized by one of the processes then the array would be cached in the *dirty exclusive* state and the benchmark would not have the intended effect. On the other hand, if the array is not initialized, then the read instruction in the for-loop body could be suppressed by the compiler, leaving only the write instruction, which again affects the event outcome. The array needs to be initialized without causing it to become cache resident, i.e., uncached, before the execution of the benchmark's main for-loop starts. Two alternatives are presented for this. The first one assumes an LRU replacement policy for

the cache. A secondary array that is bigger than the cache size is used to populate the cache after the zz array has been initialized. That is, after executing a for-loop to initialize the zz array, another for-loop is executed to write to the secondary array; this second for-loop will cause the replacement of the cache lines associated with the zz array. The only problem with this approach is that additional knowledge of the cache is needed (i.e., cache size, associativity, and replacement policy). A second alternative is to initialize the zz array and cause it to be uncached via the use the *calloc* system call, which is used to allocate its memory space. The *calloc* system call, if available on the target platform, is guaranteed to allocate memory space and set it to zeroes. Typical implementations of this system call include instructions that initialize the memory space via additional hardware (e.g., I/O instructions can map the /dev/zero file to the address space) on behalf of the CPU and, as a result, caching the array is not necessary for initialization purposes. Implementations of the *calloc* system call usually are optimal w.r.t. initializing memory space; using approaches as the one described above permits address space initialization to be done concurrently with process execution. On the target platform, platform A described in Section 3.1, the *calloc* system call approach was used.

It should be noted that this microbenchmark is performing symmetric operations on both processors. Therefore, this event should be triggered symmetrically on both processors. Assuming the behavior of one processor is replicated on the other, only one processor is monitored for validation purposes. The predicted count for the validation microbenchmark is given by the number of iterations executed in the main for-loop.

4.7.2 Data collection

As mentioned in Section 2.2, the data collection process consists of running test cases 100 times and obtaining the average test-case count. The test suite used for the requests for exclusive access to a clean cache line event includes test cases that are expected to generate from 1 to 1,000,000 events of this kind.

4.7.3 Predicted vs. hardware-reported event counts

Validation of this event was performed on the R10K platform described in Section 3.1, which supports the *clean exclusive* cache line state. As discussed above, the *calloc* system call approach was used to initialize the zz array of the validation microbenchmak. The implementation of the *calloc* system call used is the one provided by default on the IRIX 6.5 platform. Based on the results presented below, this implementation of the *calloc* system call initializes the array without caching it.

Figure 22 presents the predicted requests for exclusive access to a clean cache line event counts and the event counts reported by the hardware; the y-axis represents the percentage difference between the reported and predicted counts, and the x-axis represents the number of array references. Refer to Appendix B for the hardware-reported data.

The results indicate that the predicted and hardware-reported counts agree to within 1%. This is the case except for the test case that is intended to generate one event. In this case the standard deviation is 0.40, which indicates that there is some small overhead that is not being considered in the prediction of the event count. It is suspected that this small perturbation may be introduced by the initialization calls of PAPI itself in conjunction

with the operating environment. This is because the perturbation is not constant but it becomes insignificant as the test case gets larger. For the test case of 1,000,000 data references the difference is practically zero.

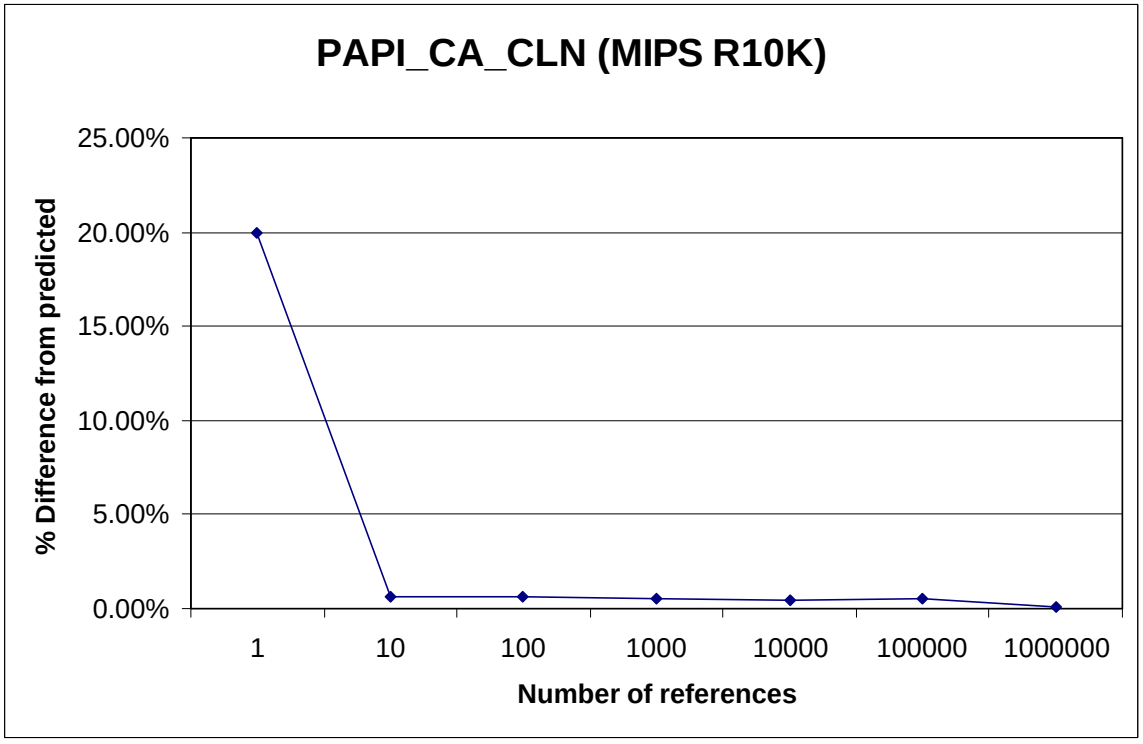


Figure 22. Requests for exclusive access to a shared cache line validation benchmark results.

Chapter 5

CONCLUSIONS AND FUTURE WORK

Efficient tools that allow the monitoring of application performance are in demand by the high-performance computing community. Performance counters provide a way to monitor the behavior of the microarchitecture without intrusively affecting its performance. Although most modern computing platforms include some sort of on-chip monitoring hardware, limited documentation is usually provided in regard to its use and the meaning of the data delivered by the hardware. Furthermore, the specific implementation issues of each platform make it difficult to define a standard. The Performance API project **[12]** has introduced a reliable, periodically upgraded tool that is easy to use and provides a common front-end across platforms. PAPI is a widely accepted tool. Its use potentially can have a great impact in monitoring the performance of applications, including those that run on heterogeneous systems using GRID technology.

To facilitate the use of PAPI, research is needed to address the reliability and usefulness of the data delivered by performance counters via PAPI. This thesis presented the results of such research for a set of seven events.

5.1 Summary of results

This thesis and the previous work referenced herein address the issues of the reliability and usefulness of the data delivered by performance counters via PAPI. This clearly is an important issue considering the wide acceptance of PAPI. The following

table lists the events studied in this thesis, the platforms on which they were validated, and observations for each.

Table 3. Summary of events studied.

Event	Platform	Observations
Data TLB misses	MIPS R10000	Reported counts are very accurate; very low overhead (4%) if event granularity is sufficiently large.
	Power3	A multiplicative difference of three for large test cases.
	Itanium	A multiplicative difference of five for all test cases.
L1 data cache misses	MIPS R10000	Reported counts are very accurate; very low overhead (almost 0%) if event granularity is sufficiently large.
L2 data cache misses	MIPS R10000	Reported counts are very accurate; very low overhead (almost 0%) if event granularity is sufficiently large.
Cache line interventions	MIPS R10000	Reported counts are very accurate; very low overhead (almost 0%) if event granularity is sufficiently large.
Cache line invalidations	MIPS R10000	Reported counts are very accurate; constant overhead of approximately 2% once event granularity is sufficiently large.
Requests for exclusive access to a shared cache line	MIPS R10000	Reported counts are very accurate; practically no overhead.
Requests for exclusive access to a clean cache line	MIPS R10000	Reported counts are very accurate; constant overhead of approximately 1% once event granularity is sufficiently large.

5.2 Future work

Further work is necessary to investigate the impact of dynamic memory allocation and initialization on the data TLB miss event. Inconsistencies were observed on the Itanium and Power3 platforms when alternating between the *calloc* and *malloc* functions for memory allocation and initialization.

Also, because of time and resource limitations, the work presented in this thesis focused only on three platforms – the IBM Power3, MIPS R10000, and Intel Itanium processors. All events except the data TLB miss event were validated for only one platform, the Origin 2000, which is based on the MIPS R10000 processor. Nonetheless, the microbenchmarks and tools developed as part of this thesis will facilitate the validation of the studied set of events on other platforms that support them. Processors for this future work are those of interest to the Department of Defense, the sponsor of this research: the Pentium microprocessor and new generations of the architectures studied, the MIPS R12000, the Power4, and the Itanium 2. These are or are becoming widely available, and PAPI support is either available or under way. Finally, the work presented in this thesis forms the basis for validating other multiprocessor events supported by PAPI.

REFERENCES

[1] S. Browne, J. Dongarra, N. Garner, K. London and P. Mucci, "A scalable cross-platform infrastructure for application performance tuning using hardware counters," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing.* November 2000.

[2] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour and T. Spencer, "End-user tools for application performance analysis using hardware counters," in *International Conference on Parallel and Distributed Computing Systems.* Dallas, TX, August 2001.

[3] M. Maxwell, P. Teller, L. Salayandia and S. Moore, "Accuracy of performance monitoring hardware," in *Proceedings of the 2002 Los Alamos Computer Science Institute Symposium*. October 2002.

[4] M. Maxwell, S. Moore and P. Teller, "Efficiency and accuracy issues for sampling vs. counting modes of performance monitoring hardware," in *Proceedings of the DoD High Performance Computing Modernization Program's User Group Conference*, June 2002.

[5] W. Korn, P. Teller and G. Castillo, "Just how accurate are performance counters?" in *Proceedings of the 20th IEEE International Performance, Computing, and Communications Conference*. Phoenix, Arizona, April 2001.

[6] M. Maxwell, "Understanding microprocessor performance event counts," M.S. thesis, University of Texas at El Paso, El Paso, TX, in progress (expected completion date: December 2002).

[7] MIPS Technologies Technical Staff, *MIPS R10000 Microprocessor User's Manual*, Version 2.0. MIPS Technologies, Inc., 1996.

[8] Silicon Graphics, Inc., "Definition of MIPS R10000 Performance Counters," September1997, http://www.sgi.com/processors/r10k/performance.html.

[9] Intel Technical Staff, *Intel® Itanium™ Processor Reference Manual for Software Optimization*. Document Number: 245473-003. Intel Corporation, November 2001.

[10] Intel Technical Staff, *Intel® Itanium™ Architecture Software Developer's Manual*, Volume 2: System Architecture, rev 2.0., Document number: 245318-003, Intel Corporation, December 2001.

[11] International Technical Support Organization, RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide, October 1998.

[12] Innovative Computing Laboratory, University of Tennessee at Knoxville, "The Performance Application Programming Interface," December 2002, http://icl.cs.utk.edu/projects/papi.

[13] J. Hennessy and D. Patterson, "*Computer Architecture, a Quantitative Approach*," Morgan and Kaufmann, 2002.

[14] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers," in *International Conference on Computer Architecture*, ACM press, pages 388-397. Barcelona, Spain, 1998.

[15] R. Saavedra and A.J. Smith, "Measuring cache and TLB performance and their effect on benchmark runtimes," in *IEEE Transactions on Computers*, 44:10, October 1995.

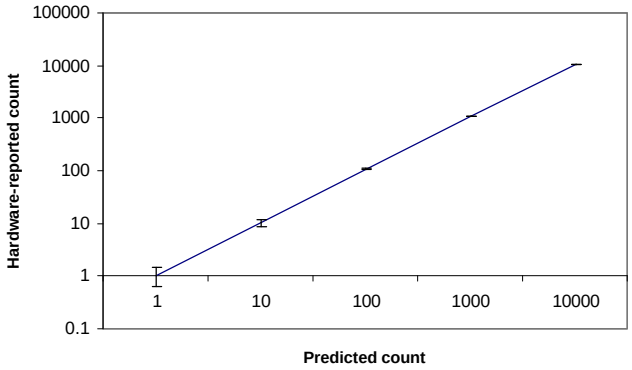
APPENDIX A: PAPI INSTRUMENTATION CODE

```
/************************************************
Leonardo Salayandia, leonardo@cs.utep.edu
PCAT research group
Computer Science Department
University of Texas at El Paso
--- PAPI instrumentation code
************************************************/
#include "papi.h"
#include "papiStdEventDefs.h"
#include "tests/test_utils.h"
int main(int argc, char *argv[]) {
        int EventSet = PAPI_NULL;
        long long **count;
        /***** Set up PAPI *****/
        if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT) {
                printf("Failed to initialize PAPI library... Invalid PAPI version.\n");
                exit(1);
        }
        if (PAPI_create_eventset(&EventSet) != PAPI_OK) {
                printf("Failed to create event set.\n");
```

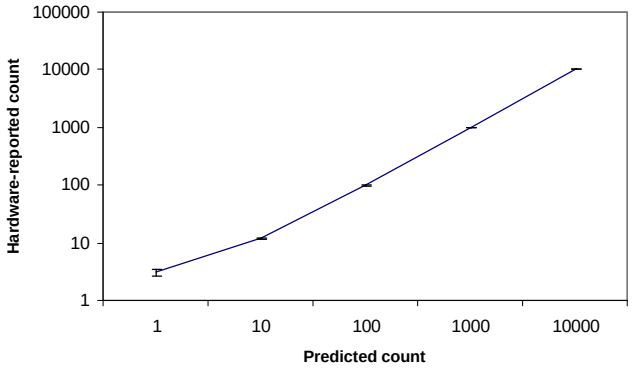
```
        exit(1);
    }
    if (PAPI_add_event(&EventSet, PAPI_EVENT) != PAPI_OK) {
        printf("Error adding event to EventSet\n");
        exit(1);
    }
    count = allocate_test_space(NUM_TESTS, NUM_EVENTS);
    if (PAPI_start(EventSet) != PAPI_OK) { /***** Start counting *****/
        printf("Failed to start PAPI.\n");
        exit(1);
    }
    /*****************************
    Benchmark core code here
    *****************************/
    if (PAPI_stop(EventSet, count[0]) != PAPI_OK) { /***** Stop counting *****/
        printf("Failed to stop PAPI.\n");
        exit(1);
    }
    printf ("%lld\n", count[0][0]); /* Print count */
    return (0);
}
```

APPENDIX B: HARDWARE-REPORTED DATA

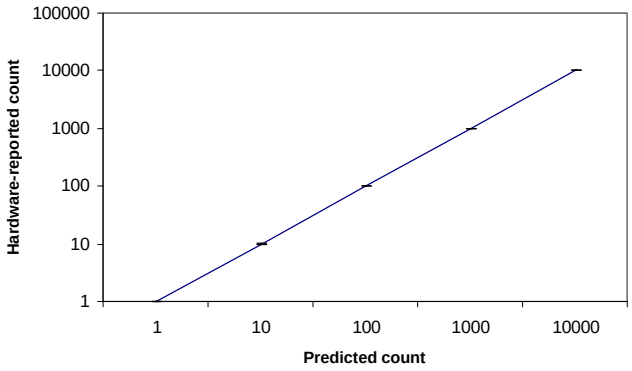
Data TLB misses (R10K)					
Predicted count	1	10	100	1000	10000
Hw-reported count (avg of 100)	1.03	10.49	108.38	1059.97	10628.73
Standard deviation	0.41	1.53	2.07	1.20	6.82



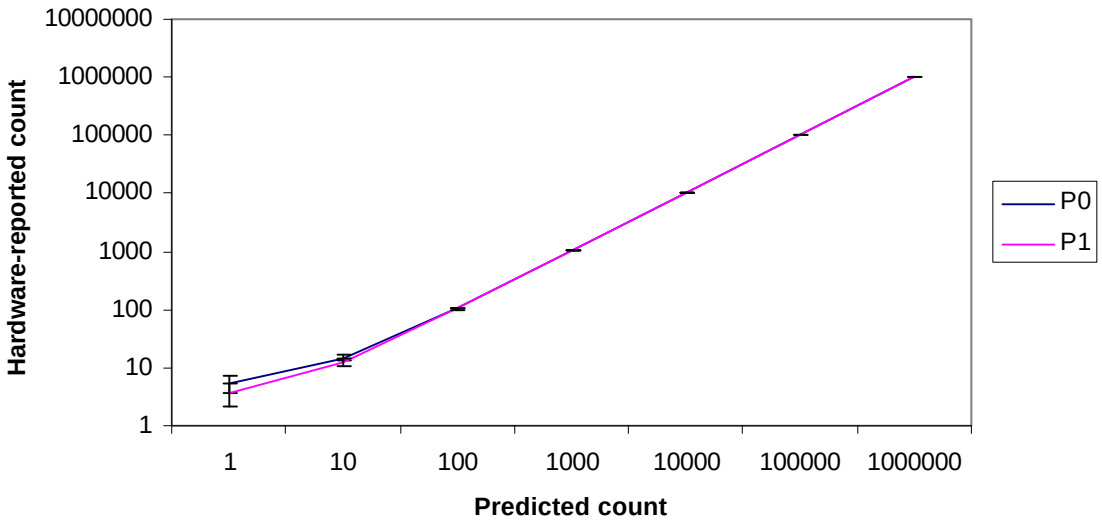
Data TLB misses (Power3)					
Predicted count	1	10	100	1000	10000
Hw-reported count (avg of 100)	3.08	12.04	99.75	1009.12	10027.00
Standard deviation	0.44	0.28	2.89	2.53	7.55



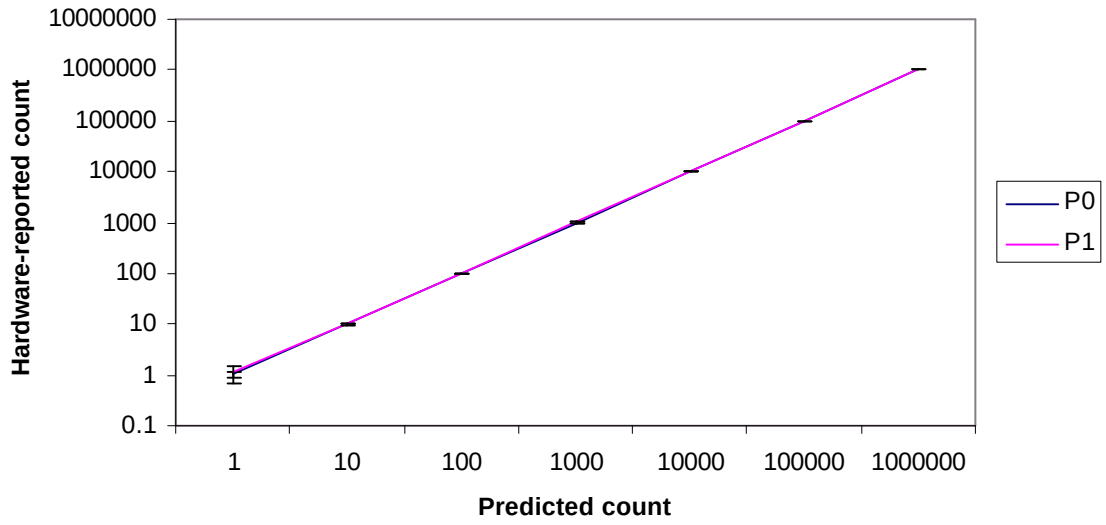
Data TLB misses (Itanium)					
Predicted count	1	10	100	1000	10000
Hw-reported count (avg of 100)	1	10.05	103.22	1004.1	10007.45
Standard deviation	0	0.26	0.54	0.44	1.09



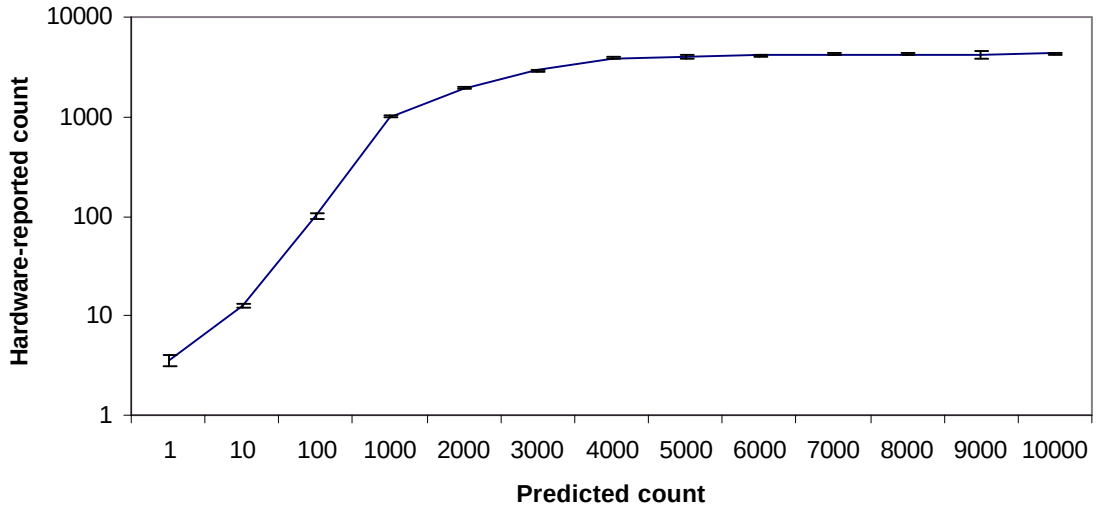
L1 data cache misses (R10K)							
Predicted count	1	10	100	1000	10000	100000	1000000
Processor 0 (avg of 100)	5.41	14.96	106.35	1026.53	10028.02	100059.2	1000391
Std dev (P0)	1.61	1.43	1.40	1.59	1.46	1.81	2.61
Processor 1 (avg of 100)	3.72	12.33	102.34	1022.54	10026.93	100053.6	1000331
Std dev (P1)	1.58	1.65	1.59	1.76	1.49	1.88	2.34



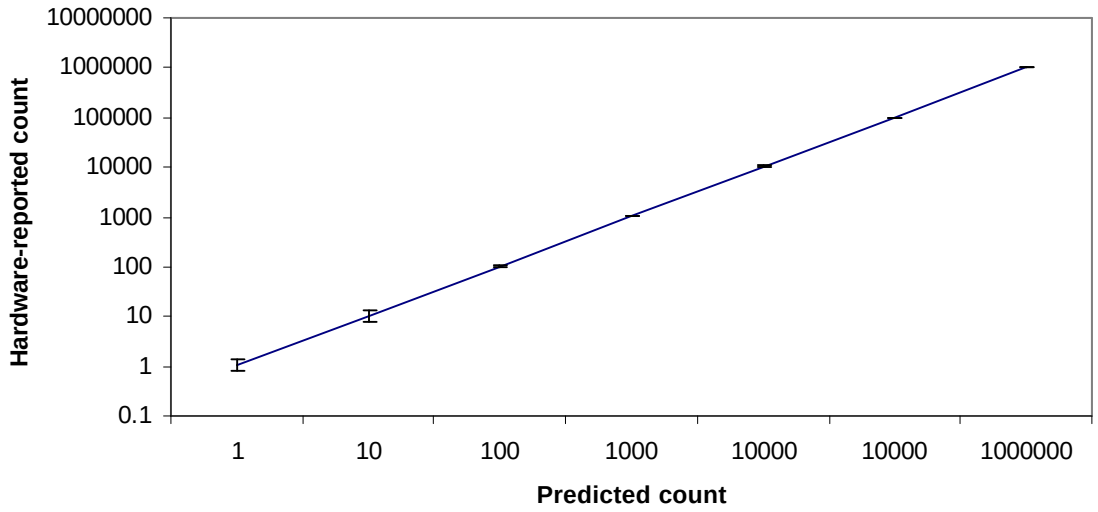
L2 data cache misses (R10K)							
Predicted count	1	10	100	1000	10000	100000	1000000
Processor 0 (avg of 100)	1.03	10.05	100.03	999.82	10020.11	100058.9	1000454
Std dev (P0)	0.17	0.22	0.17	0.50	1.54	2.74	20.47
Processor 1 (avg of 100)	1.11	10.08	100.13	1000.87	10020.11	100059.7	1000454
Std dev (P1)	0.48	0.39	0.48	1.08	1.39	1.45	7.82



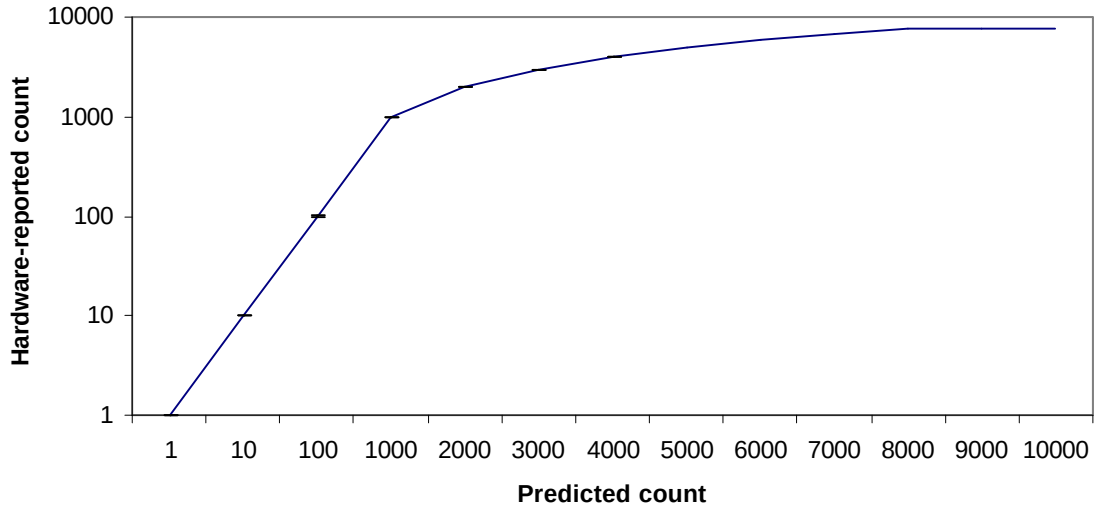
Cache intervention requests (R10K)							
Predicted count	1	10	100	1000	2000	3000	4000
Hw-reported count (avg of 100)	3.59	12.48	100.56	992.29	1938.07	2897.53	3873.69
Standard deviation	0.49	0.54	6.87	23.39	54.48	90.63	64.19
Predicted count	5000	6000	7000	8000	9000	10000	
Hw-reported count (avg of 100)	4014.22	4097.07	4213.66	4266.6	4225	4278.26	
Standard deviation	221.40	104.00	113.96	101.84	336.69	113.36	



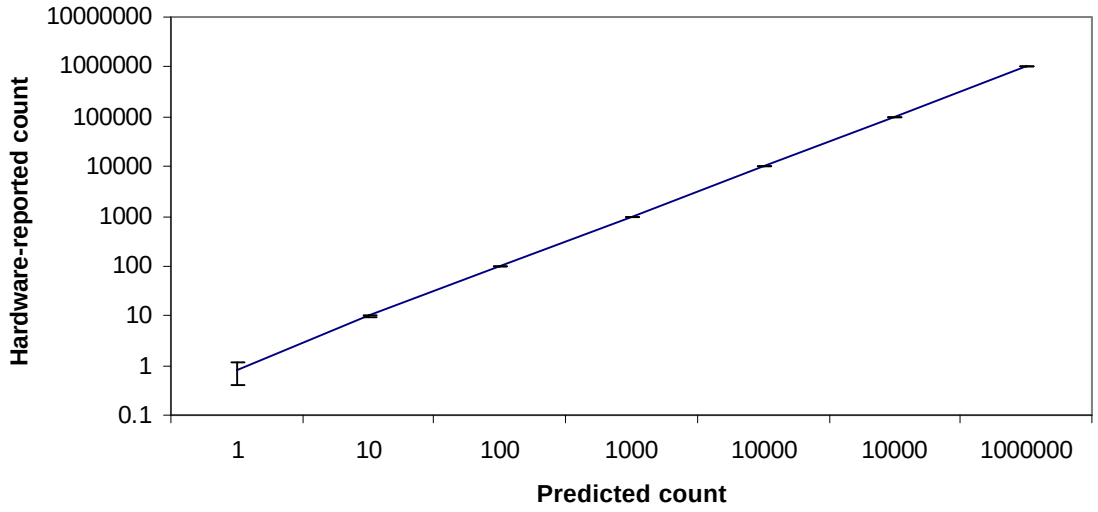
Cache invalidation requests (R10K)							
Predicted count	1	10	100	1000	10000	100000	1000000
Hw-reported (avg of 100)	1.1	10.41	101.3	1010.68	10146.8	100769.9	1006889
Std dev	0.30	2.72	4.00	6.67	419.95	485.63	1155.88



Requests for a shared cache line (R10K)							
Predicted count	1	10	100	1000	2000	3000	4000
Hw-reported count (avg of 100)	1	10	99.8	998.23	1995.64	2996.94	3984.46
Standard deviation	0	0	1.09	6.17	12.09	9.06	37.24
Predicted count	5000	6000	7000	8000	9000	10000	
Hw-reported count (avg of 100)	4922.73	5837.34	6757.27	7646.62	7840.27	7832.4	
Standard deviation	32.87	45.41	105.94	155.80	125.00	180.40	



Requests for a clean cache line (R10K)							
Predicted count	1	10	100	1000	10000	100000	1000000
Hw-reported (avg of 100)	0.8	9.94	99.34	994.7	9958.66	99482.32	998990
Std dev	0.40	0.34	1.00	3.29	36.59	1204.33	1757.33



APPENDIX C: BENCHMARK CODE

```
/************************************************

Leonardo Salayandia, leonardo@cs.utep.edu

PCAT research group

Computer Science Department

University of Texas at El Paso

--- Padding / DTLB-miss microbenchmarks

Compiled with the command line:

gcc <<C src>> -o <<OUT file>> -I/papi/src do_loops.o test_utils.o libpapi.a -O0

************************************************/

#include "papi.h"

#include "papiStdEventDefs.h"

#include "tests/test_utils.h"

#include <memory.h>

#include <malloc.h>

#include <sys/types.h>

#define NUM_TESTS 1

#define NUM_EVENTS 1

int main(int argc, char *argv[]){

int flag; /* Determines which benchmark to run, 1 = Padding, 2 = DTLB miss */

        int i, initindex, stride, max, elemXpage;
```

```
        int padsize, pagesize;
        long zsize;
        int *zz;  /* Array of integers, subject of benchmarks */
        if (argc == 5) {
                flag = atoi(argv[1]);
                padsize = atoi(argv[2]);
                pagesize = atoi(argv[3]);
                if (flag == 1)
                        zsize = (pagesize*atoi(argv[4])*2 + padsize)/sizeof(int);
                else
                        zsize = (pagesize*atoi(argv[4]) + padsize)/sizeof(int);
        }
        else {
                printf("Usage: %s <<flag>> <<pad size>> <<page size>> <<num
misses>>\nFlag = 1 for padding, 2 for validation. Sizes in bytes\n", argv[0]);
                exit(1);
        }
        zz = (int *)calloc (zsize, sizeof(int));
/*****************************************
PAPI setup code here, PAPI_EVENT == PAPI_TLB_TL or PAPI_TLB_DM
*****************************************/
        max = zsize; /* Num of elements in z array */
```

```
elemXpage = pagesize/sizeof(int); /* Num of elements in one page */
initindex = 1+(padsize/sizeof(int)); /* Num of elements in pad segment + 1 */
/* padding configuration microbenchmark */
if (flag == 1) {
        stride = elemXpage*2;
        if (PAPI_start(EventSet) != PAPI_OK) {
                printf("Failed to start PAPI.\n");
                exit(1);
        }
        for (i=initindex; i<max; i+=stride) {
                zz[i]=12;       /* Access first element on a page */
                zz[i+elemXpage-1]=12; /* Access last element on a page */
        }
        if (PAPI_stop(EventSet, values[0]) != PAPI_OK) {
                printf("\nFailed to stop PAPI.\n");
                exit(1);
        }
}
/* DTLB miss validation microbenchmark */
else {
        stride = elemXpage;
        if (PAPI_start(EventSet) != PAPI_OK) {
```

```
                printf("\nFailed to start PAPI.\n");
                exit(1);
        }
        for (i=initindex; i<max; i+=stride)
                zz[i]=12;       /* Access first element on a page */
        if (PAPI_stop(EventSet, values[0]) != PAPI_OK) {
                printf("\nFailed to stop PAPI.\n");
                exit(1);
        }
    }
    /* Print count result */
    printf ("%lld\n",values[0][0]);
    free(zz);
    return (0);
}
```

```
/*************************************************

Leonardo Salayandia, leonardo@cs.utep.edu

PCAT research group

Computer Science Department

University of Texas at El Paso

--- L1 / L2 Dcache miss validation microbenchmarks

Compiled with the command line:

gcc <<C src>> -o <<OUT file>> -I/papi/src do_loops.o test_utils.o libpapi.a -O0

*************************************************/

#include "papi.h"

#include "papiStdEventDefs.h"

#include "tests/test_utils.h"

#include <memory.h>

#include <malloc.h>

#include <sys/types.h>

#include <sys/types.h>

#include <sys/sysmp.h>

#include <sys/sysinfo.h>

#include <ulocks.h>

#include <task.h>

#define NUM_TESTS 1

#define NUM_EVENTS 1
```

```
#define CACHE_LINE_SIZE 32 /* Set appropriate cache line size in bytes */
#define OFFSET CACHE_LINE_SIZE/sizeof(int)
#define NUM_PROCS 2
void run(void);
int num_iter, mon_proc, *zz;
int main(int argc, char *argv[]){
        if (argc == 3) {
                num_iter = atoi(argv[1]);
                mon_proc = atoi(argv[2]);
                zz = (int *) malloc (NUM_PROCS * CACHE_LINE_SIZE * num_iter);
        }
        else {
                printf("Usage: %s <<num iter>> <<proc to monitor (0 or 1)\n", argv[0]);
                exit(1);
        }
        m_set_procs(NUM_PROCS);
        if (m_fork(run) == -1) {
                printf ("ERROR: Could not create child processes\n");
                exit(1);
        }
        free(zz);
        return (0);
```

```
}
void run(void) {
        int i, index, myid, temp;
        myid = m_get_myid();
        if (sysmp(MP_MUSTRUN, myid) == -1) { /* Assign process to a  processor */
                printf("ERROR: Could not assign processor to process\n");
                exit(1);
        }
        if (myid==mon_proc) {
/****************************************
PAPI setup code here, PAPI_EVENT == PAPI_L1_DCM or PAPI_L2_DCM
****************************************/
        }
        index = myid*OFFSET*num_iter;
        for (i=0; i<num_iter; i++) /* Ea processor initializes its section */
                zz[index+(OFFSET*i)] = 0;
        m_sync(); /* barrier */
        index = ((myid+1) % NUM_PROCS)*OFFSET*num_iter;
        if (myid==mon_proc)
                if (PAPI_start(EventSet) != PAPI_OK) {
                        printf("\nFailed to start PAPI.\n");
                        exit(1);
```

```
            }
        for (i=0; i<num_iter; i++)
            temp += zz[index+(OFFSET*i)]; /* Miss generated */
        if (myid==mon_proc) {
            if (PAPI_stop(EventSet, count[0]) != PAPI_OK) {
                printf("\nFailed to stop PAPI.\n");
                exit(1);
            }
            /* Print count results */
            printf ("%lld\n",count[0][0]);
        }
}
```

```
/*************************************************
Leonardo Salayandia, leonardo@cs.utep.edu
PCAT research group
Computer Science Department
University of Texas at El Paso
--- Cache intervention request validation microbenchmark
Compiled with the command line:
gcc <<C src>> -o <<OUT file>> -I/papi/src do_loops.o test_utils.o libpapi.a -O0
*************************************************/
#include "papi.h"
#include "papiStdEventDefs.h"
#include "tests/test_utils.h"
#include <sys/types.h>
#include <sys/sysmp.h>
#include <sys/sysinfo.h>
#include <ulocks.h>
#include <task.h>
#define NUM_TESTS 1
#define NUM_EVENTS 1
#define L2_LINE 128 /* Set appropriate cache line size in bytes */
#define OFFSET L2_LINE/sizeof(int)
#define NUM_PROCS 2
```

```
void run(void);
int num_iter, bar1, bar2, mon_proc;
int *zz;
int main(int argc, char *argv[]){
        if (argc == 3) {
                num_iter = atoi(argv[1]);
                mon_proc = atoi(argv[2]);
                zz = (int *) malloc (L2_LINE*num_iter + L2_LINE);
        }
        else {
                printf("Usage: %s <<num iter>> <<proc to monitor (0 or 1)\n", argv[0]);
                exit(1);
        }
        m_set_procs(NUM_PROCS);
        if (m_fork(run) == -1) {
                printf ("ERROR: Could not create child processes\n");
                exit(1);
        }
        free(zz);
        return (0);
```

```
}
void run(void) {
        int i, myid, temp;
        myid = m_get_myid();
        if (sysmp(MP_MUSTRUN, myid) == -1) { /* Assign process to a processor */
                printf("ERROR: Could not assign processor to process\n");
                exit(1);
        }
        if (myid==mon_proc) {
/****************************************
PAPI setup code here, PAPI_EVENT == PAPI_CA_ITV
****************************************/
                for (i=0; i<num_iter*OFFSET; i++) /* Monitored proc caches array */
                        zz[i] = 0;
                bar1 = 0;
        }
        else {
                bar2 = 0; /* Processor to cause interventions will keep track of barrier */
                        /* because interventions caused by barrier not in PAPI count */
        }
        m_sync();
        if (myid==mon_proc) {
```

```
            if (PAPI_start(EventSet) != PAPI_OK) {
                printf("Failed to start PAPI.\n");
                exit(1);
            }
            bar1 = 1; /* synchronize processes */
            while (bar2 < num_iter); /* loop until other proc causes interventions */
            if (PAPI_stop(EventSet, count[0]) != PAPI_OK) {
                printf("\nFailed to stop PAPI.\n");
                exit(1);
            }
            /* Print count results */
            printf ("%lld\n",count[0][0]-1);
        } else {
            while (bar1 == 0); /* synchronize processes */
            for (i=1; i<=num_iter; i++) {
                temp += zz[i*OFFSET]; /* generate intervention */
                bar2++;
            }
        }
}
```

```
/*************************************************
Leonardo Salayandia, leonardo@cs.utep.edu
PCAT research group
Computer Science Department
University of Texas at El Paso
--- Cache invalidation request validation microbenchmark
Compiled with the command line:
gcc <<C src>> -o <<OUT file>> -I/papi/src do_loops.o test_utils.o libpapi.a -O0
*************************************************/
#include "papi.h"
#include "papiStdEventDefs.h"
#include "tests/test_utils.h"
#include <sys/types.h>
#include <sys/sysmp.h>
#include <sys/sysinfo.h>
#include <ulocks.h>
#include <task.h>
#define NUM_TESTS 1
#define NUM_EVENTS 1
#define NUM_PROCS 2
#define X_VAL 0
#define Y_VAL -1
```

```
void run(void);
int num_iter, mon_proc;
int s = Y_VAL; /* shared variable initialized */
int main(int argc, char *argv[]){
        if (argc == 3) {
                num_iter = atoi(argv[1]);
                mon_proc = atoi(argv[2]);
        }
        else {
                printf("Usage: %s <<num iter>> <<proc to monitor (0 or 1)\n", argv[0]);
                exit(1);
        }
        m_set_procs(NUM_PROCS);
        if (m_fork(run) == -1) {
                printf ("ERROR: Could not create child processes\n");
                exit(1);
        }
        return (0);
}
void run(void) {
        int i, myid;
```

```
        myid = m_get_myid();
        if (sysmp(MP_MUSTRUN, myid) == -1) { /* Assign process to a processor */
                printf("ERROR: Could not assign processor to process\n");
                exit(1);
        }
        if (myid==mon_proc) {
/****************************************
PAPI setup code here, PAPI_EVENT == PAPI_CA_INV
****************************************/
                if (PAPI_start(EventSet) != PAPI_OK) {
                        printf("Failed to start PAPI.\n");
                        exit(1);
                }
                for (i=0; i<num_iter; i++) {
                        s = X_VAL;
                        while (s == X_VAL); /* read cache line until invalidated */
                }
                if (PAPI_stop(EventSet, count[0]) != PAPI_OK) {
                        printf("\nFailed to stop PAPI.\n");
                        exit(1);
                }
```

```
            printf ("%lld\n",count[0][0]); /* Print count results */
        } else {
            for (i=0; i<num_iter; i++) {
                while (s == Y_VAL);
                s = Y_VAL; /* Invalidation request generated */
            }
        }
}
```

```
/*************************************************
Leonardo Salayandia, leonardo@cs.utep.edu
PCAT research group
Computer Science Department
University of Texas at El Paso
--- Requests to a shared cache line validation microbenchmark
Compiled with the command line:
gcc <<C src>> -o <<OUT file>> -I/papi/src do_loops.o test_utils.o libpapi.a -O0
*************************************************/
#include "papi.h"
#include "papiStdEventDefs.h"
#include "tests/test_utils.h"
#include <sys/types.h>
#include <sys/sysmp.h>
#include <sys/sysinfo.h>
#include <ulocks.h>
#include <task.h>
#define NUM_TESTS 1
#define NUM_EVENTS 1
#define L2_LINE 128 /* Set appropriate cache line size in bytes */
#define OFFSET L2_LINE/sizeof(int)
#define NUM_PROCS 2
```

```
void run(void);
int num_iter, mon_proc;
int *zz;
int main(int argc, char *argv[]){
        if (argc == 3) {
                num_iter = atoi(argv[1]);
                mon_proc = atoi(argv[2]);
                zz = (int *) calloc (OFFSET*num_iter, sizeof(int));
        }
        else {
                printf("Usage: %s <<num iter>> <<proc to monitor (0 or 1)\n", argv[0]);
                exit(1);
        }
        m_set_procs(NUM_PROCS);
        if (m_fork(run) == -1) {
                printf ("ERROR: Could not create child processes\n");
                exit(1);
        }
        free(zz);
        return (0);
}
void run(void) {
```

```
        int i, myid, temp;
        myid = m_get_myid();
        if (sysmp(MP_MUSTRUN, myid) == -1) { /* Assign process to a  processor */
                printf("ERROR: Could not assign processor to process\n");
                exit(0);
        }
        if (myid==mon_proc) {
/****************************************
PAPI setup code here, PAPI_EVENT == PAPI_CA_SHR
****************************************/
        } else {
                for (i=0; i<num_iter; i++) /* The non-monitored proc initializes array */
                        zz[i*OFFSET] = 0;
        }
        m_sync(); /* barrier */
        if (myid==mon_proc) {
                if (PAPI_start(EventSet) != PAPI_OK) {
                        printf("\nFailed to start PAPI.\n");
                        exit(1);
                }
                for (i=0; i<num_iter; i++) {
                        temp += zz[i*OFFSET]; /* Read cache line, becomes "shared" */
```

```
                zz[i*OFFSET]++; /* Request exclusive access to shared line */
        }
        if (PAPI_stop(EventSet, count[0]) != PAPI_OK) {
                printf("\nFailed to stop PAPI.\n");
                exit(1);
        }
        /* Print count results */
        printf ("%lld\n",count[0][0]);
    }
}
```

```
/*************************************************
Leonardo Salayandia, leonardo@cs.utep.edu
PCAT research group
Computer Science Department
University of Texas at El Paso
--- Requests to a clean cache line validation microbenchmark
Compiled with the command line:
gcc <<C src>> -o <<OUT file>> -I/papi/src do_loops.o test_utils.o libpapi.a -O0
*************************************************/
#include "papi.h"
#include "papiStdEventDefs.h"
#include "tests/test_utils.h"
#include <sys/types.h>
#include <sys/sysmp.h>
#include <sys/sysinfo.h>
#include <ulocks.h>
#include <task.h>
#define NUM_TESTS 1
#define NUM_EVENTS 1
#define L2_SIZE 1048576 /* Set appropriate cache size in bytes */
#define L2_LINE 128 /* Set appropriate cache line size in bytes */
#define OFFSET L2_LINE/sizeof(int)
```

```
#define NUM_PROCS 2
void run(void);
int num_iter, mon_proc, *zz;
int main(int argc, char *argv[]){
        if (argc == 3) {
                num_iter = atoi(argv[1]);
                mon_proc = atoi(argv[2]);
        }
        else {
                printf("Usage: %s <<num iter>> <<proc to monitor (0 or 1)\n", argv[0]);
                exit(1);
        }
        zz = (int *) calloc ((L2_SIZE/sizeof(int))+(NUM_PROCS*OFFSET*num_iter),
sizeof(int));
        m_set_procs(NUM_PROCS);
        if (m_fork(run) == -1) {
                printf ("ERROR: Could not create child processes\n");
                exit(1);
        }
        free (zz);
        return (0);
}
```

```
void run(void) {
        int i, index, myid, temp;
        myid = m_get_myid();
        if (sysmp(MP_MUSTRUN, myid) == -1) { /* Assign process to a processor */
                printf("ERROR: Could not assign processor to process\n");
                exit(1);
        }
        if (myid==mon_proc) {
/*****************************************
PAPI setup code here, PAPI_EVENT == PAPI_CA_CLN
*****************************************/
        }
        index = (myid % NUM_PROCS) * OFFSET * num_iter;
        if (myid==mon_proc) {
                if (PAPI_start(EventSet) != PAPI_OK) {
                        printf("\nFailed to start PAPI.\n");
                        exit(1);
                }
        }
        for (i=0; i<num_iter; i++) {
                temp += zz[index+(OFFSET*i)]; /* Line becomes "clean exclusive" */
                zz[index+(OFFSET*i)]++; /* Line becomes "dirty exclusive" */
```

```
        }
        if (myid==mon_proc) {
                if (PAPI_stop(EventSet, count[0]) != PAPI_OK) {
                        printf("\nFailed to stop PAPI.\n");
                        exit(1);
                }
                /* Print count results */
                printf ("%lld\n",count[0][0]);
        }
}
```

CURRICULUM VITAE

Leonardo Salayandía was born in 1976 in Cd. Juárez, Chihuahua, México as the first of two children to Jesus and Socorro Salayandía. He graduated from the Centro de Bachillerato Tecnológico Industrial y de Servicios 128 (CBTIS 128) High School in Cd. Juárez, Chihuahua, in the spring of 1993 and entered The University of Texas at El Paso (UTEP) in the fall with the Programa de Asistencia Estudiantil para Mexicanos (Financial Assistance Program for Mexican Students). While pursuing a bachelor's degree in computer science, he worked in the Computer Science Department as a teaching assistant under the guidance of Dr. Dan Cooke and Dr. Raymond Bell. He was inducted into the Golden Key National Honor Society and the UPE Computer Honor Society in 1997 and joined the student chapter of the Association for Computing Machinery (ACM), where he served as President on the fall of 1998. Following his graduation, he worked in Talleres Diversificados de Juárez (TDJ) as head of the IT and Customer Service departments for three years, after which he returned to UTEP to continue his graduate studies. While pursuing his master's degree in computer science, he was employed as a teaching assistant for the Software Engineering course for one year under the supervision of Dr. Ann Q. Gates and as a research assistant for nine months under the guidance of Dr. Patricia J. Teller.

Permanent address: 7029 Manzanares
Cd. Juárez, Chihuahua, México.
C.P. 32330

This thesis was typed by Leonardo Salayandía.