iwmeter: A Real-Time and Interactive Parallel Software Monitor for iWarp

Ming-Jen Chan and Eka N. Ginting

February 18, 1994
CMU-CS-94-117

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3891

Abstract

Performance measurement and monitoring tools are indispensable for research and application developments on a parallel system. To be effective, such tools must be accurate, nonintrusive, efficient, and easy to understand. Private-memory MIMD machines present special challenges, especially due to the amount of data that must be gathered and analyzed, as well as the communication support required to manage these data.

This paper describes the *iwmeter*, a real-time and interactive parallel software monitor for MFLOPS performance. Implemented for the iWarp system, a private-memory multicomputer, the *iwmeter* is composed of three independent units: the sampling units, the collection unit, and the reporting unit. Each sampling unit samples the user's program in each processor in a random fashion. The collection unit collects the data gathered by each individual sampling unit, and the reporting unit processes and reports the data to the user. We conclude the paper with an evaluation of the *iwmeter*'s accuracy.

This research was sponsored in part by the Advanced Research Projects Agency/CSTO monitored by SPAWAR under contract N00039-93-C-0152. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Keywords: performance measurement, software monitor, instruction sampling, iWarp, parallel computer

1. Introduction

Measuring and monitoring parallel system performance has always been an interesting issue. After all, one important justification for a parallel system is to obtain high performance. Software tools to monitor the systems' performance are indispensable for research and application developments on these systems.

Debugging for performance in a parallel system is hard for several reasons. First, what kind of data is needed? One measure of performance is the number of floating point operations per second (MFLOPS). Memory access count or disk I/O access is another. Secondly, for each kind of data, there is a series of further questions that must be addressed. Who generates the data? How does the tool get the data? How does the tool present the data to the user? How intrusive is the tool? Data generations, data movements and collections, data processing and presentations, and the level of intrusiveness are several aspects of a software monitoring tool.

Currently, most performance debugging is done by measuring the MFLOPS. The measurement is done by explicitly inserting timing calls in the source program to time its execution. Provided that the number of floating point operations in the program is known a priori, this approach will give the user the average MFLOPS number. Such a priori knowledge of the number of floating point operations, however, is not always available, or if it is available, it may not be accurate. Moreover, if floating point unit utilization is the objective of the measurement, this approach is insufficient, since the floating point unit can be used for non floating point operations, and the compiler may generate such code. Lastly, this approach only reports one number: the average performance; an application may have peaks and valleys of performance, and to fine tune an application, the user needs a tool that reveals where the peaks and valleys occur.

Learning from the sequential programming world, we can alternatively annotate the programs to gather performance data that is replayed or analyzed at a later time (`pixie` is one such tool). In a parallel system, or even worse, in a massively parallel system, the amount of data to be post-processed is enormous. Moreover, these data are scattered in all participating processors, and they need to be written into one or more files accessible to the user. Such a tool would incur a significant overhead on the interconnection network of the processors to pass the data around, presumably to a front end system, as well as overhead on the network outside the parallel system and overhead on the file system to actually write the data out to files. On top of that, if implemented by annotating the program at the basic block level, as is done in `pixie`, the size of the program significantly expands (> 2 on average for `pixie`), and the performance is severely degraded. Thus, we need a tool that monitors performance in real time.

Another aspect of a software monitoring tool is the level of intrusiveness of the tool. There are several ways a software monitoring tool can be intrusive to the program being executed. Direct intrusion comes from the fact that some resources are required for the tool; at the very least, the tool occupies some space in memory, requires CPU cycles to execute, and requires communication channels and bandwidths to move and collect data. Additionally, the existence of a measurement tool may actually change the behavior of the parallel program being executed; a program with processors communicating with each other, for example, may see a different waiting pattern due to the existence of a monitoring tool.

In these days of extensive networking, it is rare to see a stand alone computer system, especially a parallel system. In fact, a parallel system may have another computer serving as its front end, and it may be connected to different kinds of network. For example, a SunOS machine serves as the front end of an iWarp system, which is also connected through a HiPPI network to other parallel computers and supercomputers. The user, meanwhile, usually sits on his workstation that also sits on the local network. A performance monitoring tool must be able to work in this networking environment, allowing the user to monitor the performance of a parallel system from his workstation.

The *iwmeter* is a real-time, parallel software monitor designed to address these concerns. It is currently implemented to measure MFLOPS performance of the iWarp system. We choose MFLOPS performance because it is an important performance number to know, it can illustrate the features of the design and implementation of the tool, and it is easy to measure. Other types of measurements that can be derived by examining the instructions being executed can be easily adapted into the *iwmeter* framework. In a very short time, we have built on top of the *iwmeter* framework the MBYTEmeter, a tool that measures the interprocessor data bandwidth across the iWarp system.

This paper describes the design, implementation, and evaluation of the *iwmeter*. Section 2 briefly describes features of the iWarp system that are relevant to this paper. Section 3 explains the design requirements of the tool. In Section 4 and 5, we discuss more details about the architecture of the *iwmeter* and various performance monitoring and implementation issues. Section 6 describes the evaluation of the tool, and we conclude this paper in Section 7. Appendix A provides an example of how to use this tool, and appendix B shows a screen dump of the user interface of the *iwmeter*.

2. The iWarp system

The iWarp component has been previously described in [2]. In this section we briefly recalls aspects of the system that are relevant to this paper.

The iWarp macroarchitecture supports two kinds of instructions: short and long (called the C&A instructions). A short floating point add or multiply takes 2 clock cycles for single precision or 4 clock cycles for double precision. A C&A instruction contains one floating point add and one floating point multiply - each takes 2 or 4 clock cycles to execute, depending on whether it is single or double precision. The addition and the multiplication are executed concurrently. Thus, a 20 MHz iWarp processor running all single precision C&A instructions has a peak performance of 20 MFLOPS. This variation in floating point instruction times complicates the computation of the MFLOPS performance. If all floating point instructions were executed in a fix amount of time, say 1 clock cycle, then the MFLOPS number is simply the number of floating point instructions executed in one second. This is no longer sufficient. Instead, as we elaborate further in section 5.2. under the heading of sampling issues, we must now consider the contribution of each floating point instruction to the overall floating point performance of the processor.

An iWarp processor supports single-step program execution, allowing us to examine each instruction being executed. Each processor also has 2 timer registers, each with an 8 clock cycle resolution; the contents of these two registers are automatically decremented by one every 8 clock cycles. The run time system provides support for user-level event handlers, and these registers can be used to trigger an event. These registers and the user event handler mechanism allow us to interrupt the program execution at various time intervals, and to perform various computations during each interrupt. A user event handler can be written in C (C-locale event handler) or in iWarp assembly language (asm-locale). The overhead associated with servicing an interrupt using a C-locale event handler can be as high as 500 clock cycles, whereas using an asm-locale event handler, the overhead is approximately 100 clock cycles.

The iWarp processor supports multiple logical channels between adjacent nodes. These logical channels can be used as building blocks to build communication pathways, which in turn can be used as building blocks to support various communication models such as a message passing system. Additionally, there is a network of slow wires (the SRU) running at 1 MHz clock rate connecting the processors. This wealth of communication supports presents both an opportunity and a challenge. A performance measurement tool now has at its disposal a set of communication models and resources that it can use to transfer data. At the same time, however, such a tool is required to be robust and supportive of as many communication models as practically possible.

The iWarp system is organized as a torus, and this torus is connected to a Sun host machine through an SIB processor. The iWarp run time system provides support for remote procedure calls (RPC) from the host machine to the processors, allowing us to perform various operations such as depositing and examining certain memory locations on the processors.

3. Design requirements/goals

The driving requirement for a performance monitoring tool such as the *iwmeter* is nonintrusiveness. The desire to be nonintrusive affects the accuracy, the efficiency, and the usefulness of the tool.

3.1. Accuracy

A performance measurement tool is accurate if the numbers reported by the tool are the exact, actual numbers. In the case of measuring the MFLOPS of a program executing on a processor, an accurate tool reports the exact count of floating point operations per second that the processor is executing. Since an inaccurate measurement tool can do more harm than no measurement at all, the accuracy of the numbers reported is extremely important.

We can obtain exact count of floating point instructions executed if we can afford the luxury of single-stepping the program execution. The overhead incurred by this approach, however, is prohibitively expensive. Each instruction of the user program executed is accompanied by hundreds of clock cycles spent on the interrupt. This is true for both uniprocessor and multiprocessor systems. The effect of the overhead, however, is exacerbated in a multiprocessor system when processors need to communicate and synchronize with one another.

Since we are constrained by the desire to be nonintrusive, we can only afford to sample the user program, and based on the sample, we estimate the characteristics of the entire population of instructions in the user program. We can employ statistical methods to obtain the estimate as well as to measure the accuracy of our estimate.

The accuracy of our estimates depends on the sample size, which is related to the number of times we interrupt the program execution. To be more accurate, we need to interrupt more. On the other hand, the level of intrusiveness of the tool depends on the frequency we interrupt the program execution. To be less intrusive, we need fewer interrupts. The *iwmeter* must resolve this trade off, and find the balance point in which the accuracy is acceptable while maintaining nonintrusiveness.

3.2. Efficiency

A performance measurement tool occupies some space in memory, requires CPU cycles to execute, and requires communication channels and bandwidths to move and collect data. An efficient tool requires as little of these resources as possible.

On a system like the iWarp system, where the amount of memory available per processor is relatively small, the space efficiency constraint is important. The approach of annotating every basic block, for example, will add a few instructions per basic block, resulting in a significant increase in the space requirement of the user program.

The performance measurement tool must use as few communication resources as possible. Inefficient use of communication resources can prohibit the use of the tool for certain programs. For example, in the iWarp system, there are 20 communication queues per processor, 4 of which are dedicated to the run time system. If the tool requires 2 queues per processor, the user program is left with 14 queues. Certain programs built on top of certain communication models (e.g. hypercube) cannot run using only 14 queues.

Additionally, the usage of communication resources is closely related to the communication model used to support the tool. This affects the overall overhead of the tool on the executing program. The total amount of overhead is also related to the number of times the tool can interrupt the executing program, thus affecting the accuracy of the measurement.

The tool must require as few compute cycles as possible. This requirement differs from the communication resource efficiency requirement in that inefficiency here does not result in the monitor not being usable for some programs. Instead, inefficient implementation of a performance monitoring tool results in higher overhead, which in turn affects the number of times the tool can interrupt the program, and thus the accuracy of the measurement.

3.3. Usefulness

A good tool is a tool that gets used. To be useful, there are several characteristics that are desirable in a performance measurement tool:

3.3.1. Robustness

A performance monitoring tool must support all programming models supported in the system. For example, the iWarp system supports various communication models, each with its own performance characteristics and resource requirements. It is desirable that the performance monitoring tool be capable of working with any program, regardless of which communication model the program uses. It is also desirable that the tool be compiler independent. The tool should be able to measure the performance of a program that is compiled with the iWarp C compiler as well as that compiled using the iWarp Fortran compiler.

3.3.2. Ease of use and understanding

A performance monitoring tool is easy to use if it requires the user to make no or very little modifications to the user's program. Ideally, the user should be able to monitor the execution of any program that has been compiled for the system. This, however, requires that some part of the tool be part of the operating system/run time system, because the tool may need to interrupt execution while the application is making a system call. If the user must modify the program to be measured, it is desirable that the modification be simple and straightforward.

As we mentioned in the introduction, the amount of data that a performance measurement tool in a parallel system must process and present to the user can be enormous. If each processor gathers the performance information of the program it is executing, then this information, from all participating processors, must be collected by some part of the tool to be presented to the user. The tool must be able to select, or to allow the user to select, only the appropriate information to be displayed to the user. These options should be presented to the user in an easy to understand manner.

Furthermore, since the user is most likely sitting in a network environment, a performance monitoring tool must also work properly in the user's network environment. It is also desirable to have the tool conform to the execution model of the user's network environment. A graphical user interface and the X Windows programming model provide a mechanism towards satisfying these last two requirements.

4. Architecture of the *iwmeter*

In this section we describe the architecture of the *iwmeter* and the reasons why we chose such a design.

4.1. Components of *iwmeter*

Logically, there are three components that comprise the *iwmeter*:

- A sampling unit (SU): This unit is responsible for sampling each processor and count the number of each kind of floating point instructions that the processor has executed. This unit is logically placed in each of the participating processors, so there will be as many sampling units as there are participating processors.

- A collection unit (CU): This unit is responsible for gathering from all processors the floating point counts that each processor has computed.

- A reporting unit (RU): This unit is responsible for reporting the performance measured to the user. This unit needs to be placed on a processor that can communicate directly both to the collection unit and to the user's workstation.

The separation of the CU from RU is necessitated by the possibility of mapping the CU and the RU into two different locations. As will be illustrated in the discussion on interaction models below, the CU may reside on an iWarp processor or the Sun front end, whereas the RU will be sitting on the user's workstation. In one of the models, this distinction also allows multiple RUs to attach to one CU, allowing multiple users monitoring the program execution without incurring additional sampling overhead.

4.2. Interaction models

Based on how closely these three units are interacting with each other, we can have the following models:

4.2.1. A tightly-coupled master/slave model.

In this model, the three units tightly interact, with one master, the RU, as the driver that triggers the CU, which in turn triggers the SU. The RU is the highest master, and the CU serves as its slave. In turn, the CU becomes the master of the SUs. Whenever it needs to update its report to the user, the RU will send a request to the CU, which in turn will send a request to trigger the SU. The SU will sample the user program and report whether it is a floating point instruction or not. The CU can issue these requests multiple times to the SU, gather the count from all SUs, and compute the individual processor and the array's performance measurement before satisfying the RU's request. Issuing multiple CU requests to satisfy one RU request enables the CU to report a measurement that is more accurate than simply issuing one request.

The main advantage of this model is that the user program only gets interrupted as often as the user wants the performance measure to be updated. A direct consequence of this benefit, however, is the loss of accuracy due to small sampling rate (the upper bound of which is determined by the overhead associated with sending the request from the CU to the SU). Additionally, the CU needs to broadcast one request per sampling, resulting in a higher communication overhead per sampling rate. A Unix signal-based implementation of the RU, which results in an overhead in the order of milliseconds, renders this approach entirely incapable of providing high enough a sampling rate to provide a reasonable accuracy.

4.2.2. A closely-coupled master/slave model.

The essential difference between this model and the previous model is that the CU is not triggered by the RU. Instead, the CU independently samples the user program and maintains its floating point operation count regardless of whether or not there is a request from the RU. It is still a slave of the RU, because it needs to service the RU's request for the floating point operation count when such a request arrives.

The most important advantage is that in this model, the collection rate is completely independent from the reporting rate. That is, this model allows for a higher sampling rate, resulting in a higher accuracy. The only disadvantage of this model is that, compared with the tightly-couple model above, now the user program gets interrupted more often by the SU.

4.2.3. A loosely-coupled master/slave model.

If we apply the method that takes us from the first model to the second model again, we can loosen the coupling of these units one more step. Instead of having the SUs triggered by the RU, this model lets the SUs become more independent. In this model, each SU independently samples the user program, without waiting for a trigger from the CU; and the CU updates its performance data by independently issuing requests to the SU, without waiting for a trigger from the RU. When a request from the RU arrives, the CU services the request with the performance data that it has been collecting.

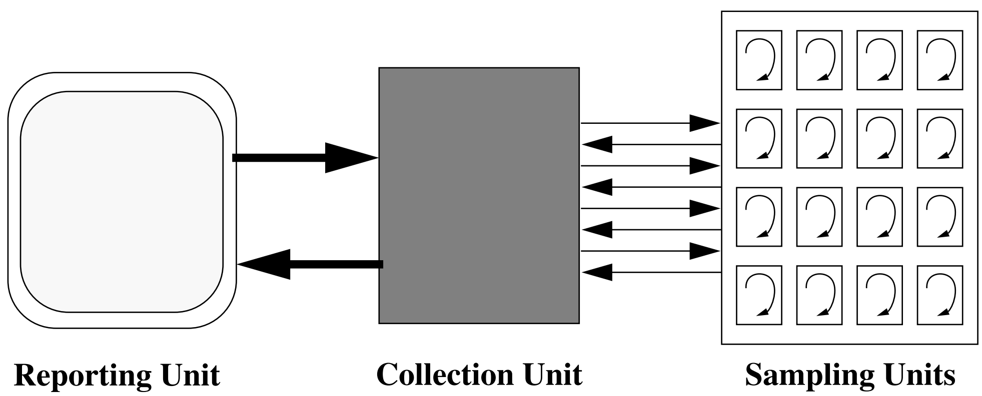

Figure 1. A loosely-coupled master/slave architecture

There are several advantages of this model. The most important one is that in this model, the sampling rate is completely independent from the collection rate. That is, this model allows for a very high sampling rate, resulting in high accuracy. The sampling rate can be as high as the equivalent of single stepping the system, and the sampling unit can proceed without incurring much communication overhead inside the parallel system. Outside of the parallel system - that is, between the parallel system (and its host machine) and the user's workstation) - communication overhead is only incurred as often as the user wants the performance measure to be updated. Furthermore, since the RU sits on a processor that can talk to both the user and the CU, this model now allows multiple RUs to connect to one CU. The disadvantage of the SU interrupting the user program more often than that of the tightly-coupled model above is still true.

This last model is superior to the other two models. Thus, the *iwmeter* is implemented based on a loosely-coupled master/slave model.

5. Performance measurement and implementation issues

This section describes various performance measurement and implementation issues.

5.1. Component implementations

The components of the *iwmeter* are implemented as follows:

- The sampling units: Each SU is a user-level program, implemented as a library, that resides in each participating processor. It is implemented using the user event-handler mechanism supported by the iWarp run time system. The initialization routine of the *iwmeter* installs the SU as an event handler, and sets the value of one of the two timer registers mentioned in section 2. The SU interrupts the execution program as follows: When the value of the register goes to zero, a timer event is raised, and the SU is invoked to handle the event. The SU samples the instruction pointed to by the program counter when the event is raised, updates the counts of floating point instructions accordingly, and resets the value of the timer register for the next sampling.

- The collection unit: The CU executes on the host machine that is connected to the iWarp system. It communicates to each of the SUs in the participating processors to obtain the floating point counts that have been executed on each processor.

- The reporting unit: The RU executes on the user's workstation that can connect through the network to the iWarp host machine. It obtains the data from the collection unit, and it processes and displays the performance measures to the user through an X-based interface.

In the current implementation, the SU categorizes each instruction being sampled based on its contribution to the floating point performance, as will be described in detail in the next section. To measure other performance characteristics of a program, instead of the MFLOPS performance, the only change required is in the way each instruction being sampled is categorized. For example, to convert the *iwmeter* into a tool that monitors register usages, the SU will need to categorize each instruction it samples based on the registers used by the instruction.

5.2. Sampling issues

As described above, each processor's program has its own SU. The SU interrupts the user program at a certain time interval, which can be statically or dynamically chosen, and examines whether the last instruction executed before the interrupt is a floating point instruction or not. We can view the sampling process as follows.

To get an exact count of floating point instructions executed, we need to single-step the user program and count how many floating point instructions are being executed. If we suppose that a processor executes the sequence of instructions $S = <s_0, s_1, ..., s_N>$ in one second, we can partition S into equivalence classes based on certain criteria on the instructions, such as each instruction's contribution to the MFLOPS performance of the processor. To measure the floating point performance of an iWarp processor, we need three equivalence classes:

1. instructions that contribute zero floating point operation per clock cycle (call it M0), which correspond mostly to non-floating point operations;

2. instructions that contribute one half floating point operation per clock cycle(M1), corresponding to short floating point instructions such as `fadd` and `fmul` that take 2 clock cycles each and double precision C&A instructions that execute 2 floating point operations in 4 clock cycles;

3. instructions that contribute one floating point operation per clock cycle (M2), corresponding to sin-

gle precision C&A instructions that execute 2 floating point operations in 2 clock cycles.

Note that for the sake of simplicity, we ignore instructions that contribute 1/4 of floating point operation per clock cycle or less (such as a double precision floating point add or multiply or a floating point divide). The true floating point performance of the processor is then given by the following formula:

$$\mu = w_0 \frac{t(M0)}{t(S)} + w_1 \frac{t(M1)}{t(S)} + w_2 \frac{t(M2)}{t(S)}$$

where $t(A)$ is the time spent executing the sequence of instructions A, and the weights w_i's correspond to each equivalence class's contribution to the floating point performance of the processor, i.e. w_0=0, w_1=10, and w_2=20 MFLOPS for a 20MHz iWarp system. For such a processor, if all instructions executed were short floating point instructions, the performance of the iWarp processor is 10 MFLOPS; if all of them are single precision C&A instructions, the performance is 20 MFLOPS.

Since we do not have the luxury of single-stepping the user program without severely degrading the performance of the program by one or two orders of magnitude (servicing an interrupt will cost several hundred cycles), we need to do a sampling of the instructions being executed. That is, we would like to sample a subsequence $R = <r_0, r_1, ..., r_k>$ of S, and partition the sample set into equivalence classes M_0, M_1, and M_2 using the same criteria used above to partition S, and compute the weighted average of the sample using a similar formula:

$$\bar{x} = w_0 \frac{t(M0)}{t(R)} + w_1 \frac{t(M1)}{t(R)} + w_2 \frac{t(M2)}{t(R))}$$

where the weights w's are the same as above. We use $\bar{x}$ to estimate μ.

How good is this estimate? Each of the sizes of the equivalence classes above is a random variable; it is a count whose value depends on what instruction is being executed at the time we interrupt the processor. However, these random variables are not independent; categorizing a sampled instruction into one class denies the other two. The value of our estimate is bounded below by 0 (all instructions are non-floating point instructions) and above by 20 MFLOPS (all instructions are single precision C&A instructions).

Intuitively, the more we sample, the more accurate our estimate is. Hoeffding[5] quantifies this intuition in the following inequality:

$$p(|\bar{x}-\mu| \leq \varepsilon) \geq 1-2e^{-2ke^2/(b-a)^2}$$

where `k` is the sample size, `a` and `b` are the lower and upper bounds, respectively. This inequality makes no assumption about the distribution of the random variables (i.e. no assumption about the distribution of the instructions being executed nor about the time intervals we perform the sampling). It provides us with a lower bound on the probability that our estimate $\bar{x}$ is within ε from the true mean μ.

If we want to have an estimate that has a probability of $(1-\delta)$ to be within ε from the true value, then we need a sample of size:

$$k = \frac{(b-a)^2}{2\varepsilon^2} ln\left(\frac{2}{\delta}\right)$$

The formula above implies that with a sample size of approximately 1,100 instructions per second, there is a 99% chance that the iwmeter reports an MFLOPS performance that is within 1 MFLOPS from the true performance of a program executing on a single iWarp processor. (Correspondingly, the MFLOPS performance number reported is within 64 MFLOPS of the true performance of a program running on a 64 processor iWarp system.)

Since servicing an interrupt costs up to 500 cycles (using a C-locale event handler), and the sampling routine itself takes approximately 125 cycles, the cost of this sampling is approximately 3% of the total 20 MFLOPS computational power available per processor. The following table describes the various combinations of error values, the number of samples required to obtain each of the error value, and the total cost to the user program as a percentage of the total 20 MFLOPS computational power available (each sample costs approximately 625 clock cycles):

Error (MFLOPS)	Sample Size (p = 0.95)	Total Cost (p = 0.95)	Sample Size (p = 0.99)	Total Cost (p = 0.99)
0.25	11,800	36.88%	17,000	53.12%
0.50	3,000	9.38%	4,200	13.13%
1.00	740	2.31%	1,100	3.44%

Table 1: Sample sizes and total costs for certain error and probability values.

This table suggests that we can perform sampling at a very reasonable cost and still maintains a high degree of accuracy. In fact, for the iWarp system, implementing the event-handler in assembly language (using asm-locale) instead of in C reduces the run time system interrupt handler overhead from 500 clock cycles to close to 100 clock cycles. We did not implement the event handler in asm-locale because the 3.44% total cost for a 99% confidence interval shown by the table above is sufficiently small. But if such an overhead is still too high, the event handler can be implemented using the asm-locale to sharply reduce the overhead cost.

5.3. Sampling randomness

If the SU samples the user program at fixed time interval, there is a possibility that it only samples instructions that all happen to be floating points (or not floating points) resulting in inaccurate measurement indicating a peak (or zero) MFLOPS performance. Such a measurement happens if the user program is uniform and happens to perfectly coincide with the time interval that the SU uses. Although this is very unlikely, we try to simulate randomness in the sampling to avoid this problem and at the same time to allow the use of statistical methods to measure the accuracy of our measurements. Furthermore, Hoeffding's formula on the previous page requires random sampling.

Sampling randomness can be accomplished by calling a random number generator at the end of each sampling and assign the returned random number as the time interval until we perform the next sampling. This, however, adds a few hundred clock cycles to the cost of a sampling. An alternative approach is to precompute a table of random intervals. The sampling unit cycles through the table, and sets the next sampling time to be the current time plus the content of the current entry of the table. There are two problems with this approach. The first problem is the overhead of constructing the table. This overhead, however, will be amortized over the length of time the program executes. The second problem, which is more severe, is the space required for the table. As the table above suggests, to get a 99% confidence interval, we need a sample size of approximately 1,100 instructions for each second of execution. Storing the table entry as integer requires 4.4KB of memory, if we reuse the table during the entire execution time.

Alternatively, we can simulate randomness here by varying the time interval as a function of the value of the third from the last bit of the program counter. That is, the time to do the next sampling is first initialized to a certain value. At each sampling, we either increment or decrement the time interval for the next sampling by looking at the third from the last bit of the program counter, ensuring that the value stays within the interval of one half to twice the initial value. Since a priori the user program's execution flow is unknown, and the event handler may be first invoked at any position in the user program, the probability that we either decrement or increment the initial value approximately equals the probability that the program counter's third from the last bit is either zero or one. The program counter is incremented by the size of the instruction. Since an iWarp instruction is either 4 bytes or 12 bytes, the program counter is incremented by either 4 or 12, resulting in always toggling the third from the last bit. Thus, we expect that the probability that we either decrement or increment the time interval for the next sampling is approximately one half. Section 6.1. discusses our verification of this simulation of randomness, as well as our comparison of this approach to the table-based approach above.

5.4. Communication for data collection

The collection unit (CU) must broadcast a message to the participating processors to request the processors' floating point operation counts. There are four ways to accomplish this communication:

5.4.1. SRU printf

The CU uses the slow SRU wires to broadcast a request to the processors and to receive the responses. The overhead per processor is approximately 70 ms per collection, leaving enough slack for one collection per second. This approach does not take away any of the processor's communication resources. Although it will work with programs that use various communication supports, this approach does not work with programs that use the SRU (such as those that use SRU-based barrier synchronization[3]). Furthermore, this approach requires that the CU be either on the SIB processor (which connects the iWarp processors to the Sun front end) or on one of the processors.

5.4.2. Dedicated ring (pathlib or PCS)

The *iwmeter* can require the support of a dedicated ring (formed using pathlib or PCS[4]) to broadcast collection requests and to receive the processors' responses. This approach is faster than the other approaches discussed here, and it incurs lower overhead. The disadvantages are that it requires two dedicated queues, leaving only 14 queues for the user, and that the CU still requires either the SIB or one of the processors. Further, if the ring is PCS-based, this tool can only be used for PCS programs.

5.4.3. Message passing (dmsg)

The *dmsg*[6] deposit message passing support can also be used by the CU to directly deposit the request to the processors, and the processors can directly deposit the responses to the CU. This approach is very simple to program, and the overhead of a deposit is very low (approximately 50 clock cycles). The problem is that it reserves 5 queues for communication. Similarly, it requires that the CU be either on the SIB or one of the processors.

5.4.4. Remote procedure call (RPC)

The iWarp RTS supports an RPC interface from the host processor to the processors that allows one to export the content of a certain memory location on a certain processor. Using this support, the CU can sit on the host processor, instead of requiring the SIB or one of the processors, and read the content of the memory where the SU maintains its floating point operation count. This approach uses only the RTS queues, thus does not take away any communication resources from the user. The SU is now completely independent; it does not even need to explicitly respond to the CU's request. The overhead associated with the RPC is relatively high, but is still feasible within the time window of one collection per second.

The iwmeter uses the RPC mechanism, which clearly has the advantage over the other mechanisms.

6. Evaluation

We evaluated the *iwmeter* on two grounds. First, how good is our approximation of sampling randomness? Second, how accurate are the MFLOPS performance numbers that the *iwmeter* reports?

6.1. Sampling randomness

Sampling randomness is obtained by constructing a table of random sampling intervals. This table is generated in the initialization phase by invoking a random number generator. The next sample is taken at an interval given by the next entry in the table, returning to the beginning of the table once the last entry is used. The size of the table is approximately 1,300 entries, corresponding to roughly the number of samples performed in one second.

To avoid the space cost of the table, we simulate sampling randomness by incrementing or decrementing the time interval to perform the next sample depending on whether the third from the last bit of the program counter is odd or even, expecting that the bit used has an equal probability of being odd or even. We verified this expectation by maintaining a count that gets incremented if the bit is odd and decremented if the bit is even. If the expectation is true, then the count is zero at the end of the program. Sampling test programs at approximately 1,300 times per second, we obtained the following result: the average difference in the number of odd versus even is 4.49, with a standard deviation of 0.04. That is, on average, the number of times that the third from the last bit is odd differs from the number of times it is even by less than 5 out of approximately 1,300 samples, giving a $P(odd) = 0.5015 \approx 0.4985 = P(even)$. However, the sampling intervals generated by this optimization fail to pass the chi-square and Kolmogorov-Smirnov testing of their randomness.

6.2. Measurement accuracy

To measure the accuracy of *iwmeter*, we wrote a program that generates programs with known floating point performance. Each program is mainly composed of an outer loop that takes much less than one second. Inside the loop there are two loops: one that is composed of solely C&A instructions, and one that is composed of solely non-floating point instructions. If the ratio of the C&A loop and the non-floating point loop is 0.55, then the expected performance of the program is 11 MFLOPS on each processor. We compare the expected number and the average of the reported numbers over a period of approximately 35 seconds for 41 programs with expected performance numbers ranging from 0 to 20 MFLOPS. The sample size is approximately 1,300 samples per second, giving a 99% probability that the reported performance number is within 1 MFLOPS of the true performance.

Using the optimization described above, the following figure gives, for each of the 41 programs, the average difference between the expected and the observed MFLOPS values, as well as the range of the differences:

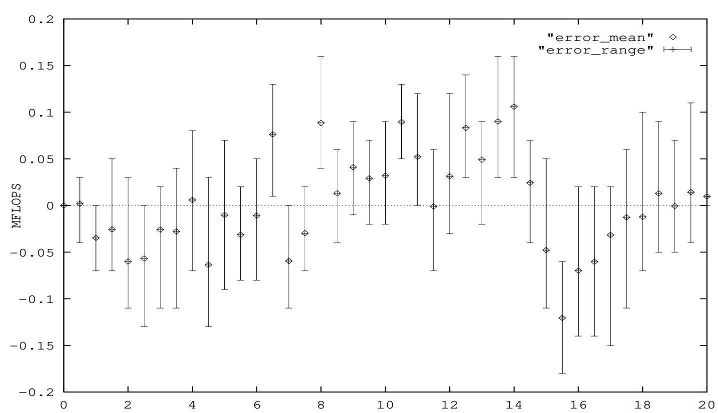

Figure 2. Error values (in MFLOPS) using the 3rd from the last bit to determine the next sampling.

The figure above shows that the tool is indeed very accurate. The mean of the errors is 0.001 MFLOPS, with a standard deviation of 0.051 MFLOPS. The ranges of errors are small, varying from 0 (when there is no floating point operation in the loop) to 0.17 MFLOPS. In all but seven cases, these ranges covered the expected MFLOPS.

Using the same 41 programs but with the sampling interval table, the ranges of the differences between the expected and the observed MFLOPS values are shown in the following figure:

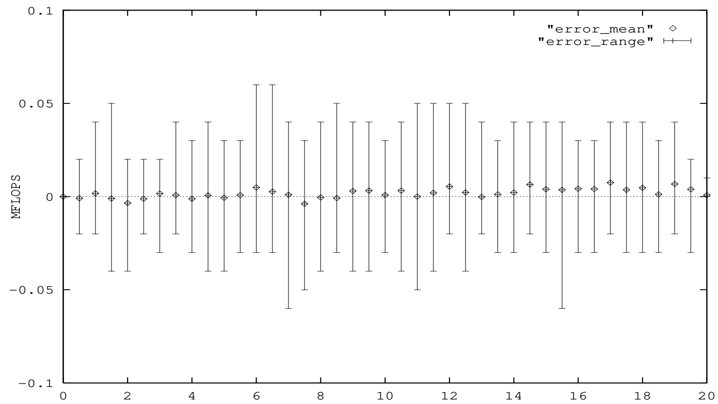

Figure 3. Error values (in MFLOPS) using a table of random sampling intervals

This figure shows that the result is even better than the previous approach. The mean of the errors is 0.0018 MFLOPS, which is only slightly greater than the previous approach, but the standard deviation is much smaller (0.0026 MFLOPS). The ranges are also much tighter, varying from 0 to only 0.0075 MFLOPS, with all ranges covering the expected MFLOPS performance.

In conclusion, the approach of using a table of randomly-generated sampling intervals provide smaller ranges of errors, while only slightly increasing the mean of the errors. Furthermore, this approach allows us to use the theoretical result discussed in the previous section to define the level of confidence of our estimate.

7. Conclusions

The need for a tool to monitor the MFLOPS performance of a program running on a parallel system like the iWarp is satisfied by a program such as the *iwmeter* described in this paper. The *iwmeter* tool has a well-defined, high accuracy. It is designed and implemented without extra communication resources (other than those required by the run time system) with a very low computation cost. It is also a general and robust performance monitor, because it allows for monitoring of all iWarp programs regardless of the communication models of the programs.

The *iwmeter* is nonintrusive; at any given second it allows for approximately 96.6% of user code execution for a sampling rate of approximately 1,100 samples per second, while still maintaining very high accuracy. The graphical user interface is X-based, resulting in an easy to understand interface. Lastly, the library support provided to use the *iwmeter* is the easiest possible without requiring full and direct support of the run time system.

The *iwmeter* is currently implemented to measure MFLOPS performance. Various other measurements can be accommodated within the framework of the tool with very little modifications. For example, if one wants to monitor memory accesses, the underlying sampling and reporting mechanisms that *the iwmeter* provides will sufficiently support such a measurement. The only change will be in the section of code that categorizes the instruction being sampled and update the appropriate counts. Thus, the *iwmeter* provides an underlying model, design, and implementation for monitoring of program characteristics so long as information about these characteristics can be gathered by examining the instructions sampled from the user code. The requirements for an effective tool for such measurements will be identical to those faced by the *iwmeter*, and thus are still satisfied by the *iwmeter*.

Bibliography

[1] Borkar, Shekhar, et.al. *iWarp MacroArchitecture Specification, Version 3.2.* Carnegie Mellon University and Intel Corporation. February 1991.

[2] Borkar, Shekhar, et. al. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings Supercomputing '88*, pages 330-339, Orlando, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.

[3] Feldmann, Anja, et. al. Barrier Synchronization on Private-Memory Machines.In *Proceedings SPAA '92*, San Diego, California, 1992.

[4] Hinrichs, Susan. *Programmed Communication Service Tool Chain User's Guide, Version 2.5.2.b*. Carnegie Mellon University. May 1992.

[5] Hoeffding, W. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association, 58 (301)*. 1963

[6] O'Hallaron, Dave. *dmsg* Source Code. 1992

Appendix A. Usage and example

The relevant files of the *iwmeter* tool can be found in /afs/cs/project/iwarp/member/eginting/iwmeter and its subdirectories.

A.1. Usage

Currently, the *iwmeter* tool is composed of a library, a collection daemon, and an X Windows program. To use the *iwmeter* tool to monitor the MFLOPS performance of the user's program, the user needs to:

A.1.1. In the user program:

- Include `<mflops.h>`
- Call `mflops_cell_init()` as an initialization procedure at the beginning of the user code
- Link with `libmflops.a`

A.1.2. To monitor the results:

- Run the *iwmeter* daemon on the SUN machine that hosts an iWarp system:

```
<user@daemon-machine>% rpc.iwmeterd <iwarp-hostname>
```

- From a user workstations, the user needs to run the program *iwmeter* to perform the monitoring:

```
<user@workstation>% iwmeter <daemon-machine-name>
```

- Run the program that has been compiled as described above.

A.2. Example

The following example is generated by the test program generator and used for verification of the *iwmeter*:

```
/*
 * Generated by gen-test.
 */
#include <stdio.h>
#include <mflops.h>
#include "asm_loop.h"

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

        mflops_cell_init();
        t = 5000;
        while (t--) {
                asm_loop();
        }
}
```

Appendix B. User interface

The figure below is a screen dump of the user interface of the *iwmeter*. The user can select whether to display the performance data of all participating processors or just the entire system summary. Future improvement may include selectively activating/deactivating only the processors that the user wants to monitor.

The user may also select the time interval the report is updated. The default is one second. Additionally, the *iwmeter* can display the performance numbers of an instant or compute the moving average of the past several numbers.

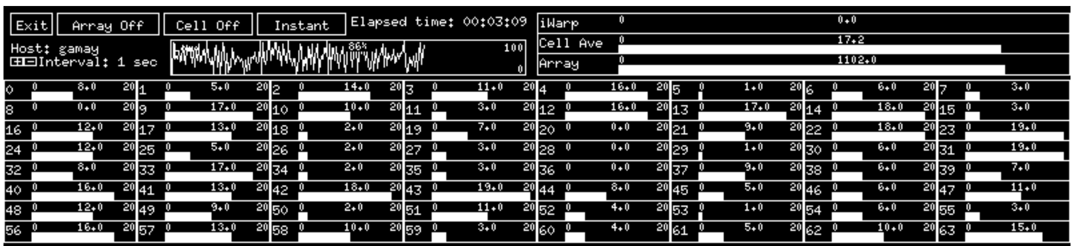

Figure 4. The iwmeter user interface