Using Hardware Counters to Improve Dynamic Compilation

ECE1724 Project Final Report

December 18, 2003.

David Tam
tamda@eecg.toronto.edu

John Wu
johncwu@ca.ibm.com

ABSTRACT

In this paper, we describe our project to explore the use of hardware counters to improve triggering techniques for run-time dynamic code recompilation. The Intel Open Runtime Platform (ORP) was chosen as the target Just In Time (JIT) compilation-capable Java Virtual Machine (JVM). The performance counter library (PCL) implemented by Rudolf Berrendorf et al. was used to retrieve real-time micro-architectural level performance values. Only one of these values, the cycle count, was used in this project. Although, the benchmark performance results were not as good as originally anticipated, the work completed for this project has created an infrastructure that can be easily reused and adopted to use other hardware counter values to create improved compilation triggering mechanisms. We introduce a different and potentially better way of performing runtime profiling in comparison to software-only technique used in the original ORP.

1. INTRODUCTION

Modern software heavily utilizes shared libraries, dynamic class loading, and run-time binding. This makes it more of a challenge to perform optimization at static compilation time. Moreover, for cross platform compatibility reasons, programs written in languages such as Java are compiled into some form of intermediate representation, such as some machine-neutral byte-code. The byte-code is then compiled into native machine code by a dynamic compiler at runtime.

Because byte-code to native code conversion happens at application execution time, it offers new opportunities to the dynamic compiler to perform sophisticated, profile-guided and architecture-specific compilation. However, these sophisticated optimizations often come with heavy overheads, which is factored into the overhead of the application's execution. Unless the performance increase exceeds the optimization overhead, such optimization may not be worth performing. Runtime environments that provide Just In Time (JIT) optimized compilation are designed to identify *hot traces*, or places where the application spends most of the time, using run-time profiling techniques, and target these *hot traces* for optimized compilation. The Intel Open Runtime Platform (ORP) is one example of such Just In Time capable environments. [8]

The ORP JVM is the open source version of the Intel Microprocessor Research Lab (MRL) Virtual Machine, as studied in class in [7]. In this paper they describe two fairly primitive and approximate techniques in determining when to trigger the optimizing compiler. However, it should be possible to provide a more accurate triggering threshold with the help of microprocessor hardware counters.

Hardware counters offer a low overhead mechanism of obtaining valuable and precise run-time performance information that can improve the task of run-time/dynamic compilation/optimization. Hardware counters can provide important micro-architectural performance information that is otherwise unobtainable from software techniques alone. Many of these parameters can potentially offer accurate real-time performance information for the purpose of run-time instrumentation.

The complete list data acquirable through the hardware counters that are available using the PCL package used for this project is shown in Table 1

L1 Cache Read, Write, ReadWrite
L1 Cache Hit, Miss
L1 Data/Instr Cache Read, Write, ReadWrite, Hit, Miss
L2 Cache Read, Write, ReadWrite, Hit, Miss
L2 Data/Instr Cache Read, Write, ReadWrite, Hit, Miss
Translation Look-aside Buffer (TLB) Hit, Miss
Instruction TLB (ITLB) Hit, Miss
Data TLB (DTLB) Hit, Miss
Cycles, Elapsed Cycles, Instructions
Integer/Floating-Point Instructions
Loads, Stores, Load/Store Instructions
Jumps, Successful, Unsuccessful
Atomic Operations, Successful, Unsuccessful
Stalls, Integer/Floating-Point/Jump/Load/Store Stalls
MFLOPS, IPC
L1/L2 Data Cache Miss Rate
MEM_FP_RATIO

Table 1: Micro-architectural statistics obtainable with PCL

Not all of these values are implemented or acquirable on all platforms, but it presents a long list of potential parameters that could be used to help in run-time code instrumentation. In this paper the use of hardware counters in one very specific scenario, the cycle count, is exploited to improve the timeliness of dynamic compilation.

2. RELATED WORK

Based on the most recent publication from the Intel ORP project [8], the triggering techniques have not been improved yet. Further literature search did not reveal any significant work with the same goal as what was proposed in our project. This situation provided an opportunity to conduct some original work.

The most recent and closest work that could be found was by Chen et al. [6]. They have done some initial investigation into hardware counters for use in dynamic optimization. However, they have not integrated it with a real system yet (such as Jalapeno/Jikes [1], JUDO [7]). The paper does only initial investigation into the idea of using hardware counters to improve dynamic compilation accuracy. In actual fact, the paper was mainly about using hardware counters to more accurately build traces. It was basically trying to re-create the HP Dynamo [4] trace-based model but with the use of hardware counters. So, they only discuss issues such as trace coverage (the number of execution cycles that was consumed within the traces) and trace building overhead. None of these issues are a concern to us compared with what was proposed and done in their paper.

Krintz and Calder [10] presented a framework (under ORP) to reduce the time spent performing optimizations. Annotations were added to guide the optimization process so that time consumed by the optimizer could be spent wisely. However, hardware counters were not used. Instead, they used an offline technique to determine where compiler optimization time was spent. Hardware counters could potentially be used to extend this annotation framework. However it was determined that this was beyond the scope of this project.

The Jalapeno JVM [1] offers a multiple optimization level scheme. Hardware counters can be used to improve the accuracy of their cost/benefit model for determining when to trigger a dynamic recompilation/optimization as well as to which optimization level to target. Unfortunately, for the purposes defined by this project, the fact that Jalapeno is written in Java would have complicated matters. So we decided not to base the project on this JVM. Native method invocations (to access hardware counters) may complicate the "low overhead" goal of hardware counters.

Arnold et al. [2] [3] have described a low-overhead software-only approach to performing online instrumentation and feedback-directed optimization but did not make use of hardware counters.

Kistler and Franz [9] describe their runtime system (Oberon) that performs continuous program optimization. A key component in their system is the profiler. The authors mention the benefits of using hardware counters in addition to software-based techniques (instrumentation and sampling) to aid in the continuous optimization task.

There are many potential uses of hardware counters since they can be used with relatively low overhead and can offer important micro-architectural performance information that is otherwise unobtainable. Such information could be fruitfully used by dynamic optimization systems such as JVMs and JITs. This project only explored one possible use of hardware counters, which was to provide more accurate profiling information for the purpose of triggering dynamic optimizing compilation.

3. DETAILS

3.1 Tools and Platform

A base set of tools and platform were selected for this project. The list includes a performance counter API library, a target JIT capable JVM that is profiling based, and the hardware and operating system. The reasons for selecting this particular set of tools and platform are listed below.

3.1.1 PCL

It was determined that in order to achieve simplicity and portability in the implementation, it was wise to choose a mature performance counter application programming interface library, instead of implementing it purely in assembly. As well, time-constraints were a limiting factor and we did not want to learn the intricacies of x86 assembly coding.

A number of performance counter library solutions were evaluated. The first two were considered were the IBM AIX Performance Monitor API Library, which is a collection of function calls implemented in C, and another unsupported set of assembly routines implemented by a group at IBM T.J. Watson Research Center. Both libraries were created to work with the IBM PowerPC platform. In order to use this library, a PowerPC compatible JIT JVM was required. Jalapeno or Jikes would have been the only choices. However, both Jalapeno and Jikes were implemented in pure Java. As mentioned in the Section 2, this meant making native method invocations, where may have high overheads, or implementing a Java port of the IBM AIX Performance Monitor APIs Library or a Java port of the unsupported assembly routines need to be implemented, which would increase the risk factor of this project.

The Performance Counter Library (PCL) developed by Rudolf Berrendorf et al. [5] was the next on the list. The library was also implemented as C like functions, and targets a number of hardware and software systems including Linux on Intel Pentium, Pentium MMX, Pentium Pro, Pentium II, Pentium III, Pentium 4, AMD Athlon, Duron, AIX on IBM Power PC, Tru64 and CRAY Unicos on DEC Alpha, SGI IRIX on SGI R10000, R12000, Solaris on UltraSPARC I/II/III. Even though it is still implemented in C, it targets many more hardware software systems, which in turn makes the list of potential JIT JVMs longer. Most importantly, the PCL is open source and can be freely redistributed.

After careful consideration, the PCL was chosen as the performance counter instrument tool for this project.

3.1.2 ORP

Because the PCL was chosen as the performance counter instrumenting tool for this project, it was necessary to choose an open source JIT JVM that was implemented in C/C++. ORP naturally became a potential candidate. ORP and its associated JIT compilers were implemented on the IA32 platform for both Linux and Windows NT. Finally, the fact that we know ORP reasonably well through the JUDO paper made ORP our final choice as the target JIT JVM.

3.1.3 Linux and GCC

Although ORP runs on both Windows NT and Linux, the PCL chosen does not run on Windows NT, but runs on Linux. Linux became the only choice for our operating system at this point. Redhat Linux with kernel version 2.4.18.x was chosen. The particular kernel version was chosen because the PCL implementation for Linux required the application of a special kernel patch, which was only compatible with a selected number of kernel versions.

GCC 3.2.1 was chosen to compile the kernel patch and to recompile the Linux kernel, as that was what was required. GCC 2.96 was used as the main compiler for this project, as it was required to compile ORP and the PCL library on Linux.

3.1.4 Hardware

A computer with two AMD Athlon processors running at 1.5 GHz (1.8 GHz equivalent) was used. Each processor had 256KB of L2 Cache, and the computer had 512MB of memory. There was no particular reason for choosing this computer other than this was the one that had root access available so that the Linux kernel could be recompiled.

3.2 Method Profiling

Our task of instrumenting the O1 code was more involved than simply instrumenting method entry points, as was done in [7]. The increased complexity was due to the need to find closure for every *start cycle* point. That is, there must be corresponding *stop cycle* points so that *cycle deltas* can be obtained and accumulated for the appropriate method. This need to maintain extra state and account for all *escape* points was not present in [7].

For single-threaded applications, the following scenarios must be considered. (1) Start and end of methods. There can be multiple end points due to multiple `return;` points. (2) Method invocations. At method invocation sites, the cycle counter of the caller must be stopped. At the subsequent line after the method invocation site, the cycle counter of the caller must resumed because this is the return point from the callee.

Figure 1 illustrates a simple scenario involving two methods/functions. The cycle counter value is obtained from the hardware counter at the beginning of function A and noted in function A's profile record. Upon invocation of function B, the cycle value is again obtained and the delta is added to function A's *accumulated cycles* profile record. Upon return from function B, the hardware cycle counter is read again. Finally, upon exit of function A, the cycle value is again obtained and the delta is added to function A's *accumulated cycles* profile record.

The above implementation accounts for single-threaded applications. Multi-threaded application support has not been implemented due to project time constraints. In addition, the impact of parallel/threaded garbage collection has not been accounted for. Conceptually, accounting for these two environments is simple. For multi-threaded support, upon thread switching, the *start/stop* functions indicated in Figure 1 can be called by the thread scheduler to start and stop the cycle counter for the appropriate threads. Statically inserting these lines into the thread scheduler can be considered much easier than dynamically inserting x86 assembly instructions into O1 code. The ORP thread scheduler is written in C/C++ and we would be statically adding C code. To handle threaded garbage collection, the first thing the garbage collector would do is to stop the cycle counter of the current application thread. Upon exit of the garbage collector, it would resume the cycle counter of the interrupted thread. In fact, adding multi-threaded support could potentially automatically handle the garbage collector since it may appear as a regular thread to the thread scheduler.

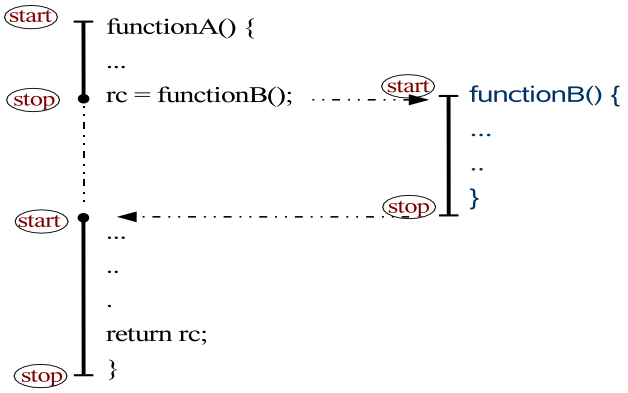

Figure 1: Profiling scenario

These *start* and *stop* instructions were inserted into the O1 code as the ORP JVM lazily translates Java byte-code to O1-level x86 assembly code. Since our inserted pieces of code must be in assembly, we decided to insert a simple x86 `call` instruction to jump to our start/stop counter functions, which we wrote in C/C++. In that way, we could write simple, high-level C code and make use of the PCL library API rather than figuring out how write all of this in assembly. We recognize that there could be a performance trade-off because of the need to branch to our start/stop functions before executing the invoked method, however, our results in Section 4.2.2 show that the overhead is negligible.

3.2.1 Profiling O1 Code

The start and stop functions required the address of a profile record to the passed to it as a parameter. This information is required so that the start or stop cycle information can be billed to the correct method. This parameter passing was accomplished by inserting x86 assembly code to push the appropriate address onto the stack before the O1 code calls the start/stop function. When the start/stop function is called, it automatically pops the stack to obtain the passed parameter.

Instructions to perform code insertion were added to the `emit_prolog()` and `emit_epilog()` functions of the O1 JIT compiler. These two functions covered the *start* and *stop* annotations that are necessary in the entrance and exit of methods. To handle the scenario where a method invokes another method (and returns from the invoked method), we had two possible implementations in mind.

1. We could insert x86 code to the current method *before* it invoked target method and in the subsequent line *after* the method invocation site. This would involve adding code to the following functions of the lazy code generator so that they insert the desired x86 code into the O1 code: `gen_invokevirtual()`, `gen_invokestatic()`, `gen_invokespecial()`, `gen_invokeinterface()`, and `gen_get_method_return()`.
2. We could add C code to our existing start/stop functions to pause and resume the caller's cycle counter.

We shall refer to (1) as *design decision A* and (2) as *design decision B*.

We attempted to implement the functionality using both approaches and found *design decision B* to be successful. We did not have adequate time to continue attempting *design decision A* so we used *design decision B*. In terms of implementation complexity, *design decision B* required a special *profile record* stack to be implemented to keep track of callers. Perhaps the most significant implementation disadvantage was that this had to be implemented for both the O1 JIT and O3 JIT. This was necessary because and O1 method may invoke an O3 method. The O3 method must access the *profile record* stack and pause the O1 caller's cycle counter.

In terms of performance trade-offs, we believe that *design decision A* would perform better since it would make use of local data before invoking the target method. Consequently, it may improve temporal cache locality slightly compared *design decision B*. As well, through our experimental results, we believe that *design decision B* is detrimental to O3 code. Since O3 code is highly optimized and does not keep track of cycle counts for its methods, the O3 code must perform extra work that will slow down its execution. The extra work is quite a lot for the O3 code. It must call the start/stop function of the caller method, subsequently make a call to the PCL library.

3.2.2 Profiling O3 Code

We did not plan to instrument the O3 code because there were no higher optimization levels to ascend to. However, due to the use of *design decision B*, it was necessary to insert start/stop function calls into O3 code.

The O3 JIT compiler had a completely different way of inserting code than the O1 compiler. This required learning a completely new method, which as fairly time-consuming. It basically involved adding code to the `insert_prolog()` and `insert_epilog()` functions of the O3 code emitter so that it would insert calls to our start/stop functions. The start/stop functions were slightly different from the O1 start/stop functions. We shall refer to them as the O3 start/stop functions. In the O3 versions, cycle counter information for the O3 method is not tracked. however, pausing of the caller's cycle counter is done. As well, a *dummy* profile record is pushed onto the *profile record* stack because the O3 method may call an O1 method, which assumes normal operation of the profile record stack.

The O3 start/stop functions did not require a parameter to be passed to it, since the O3 method profile records are not used. As a consolation prize for using *design decision B* the elimination of parameter pushing and popping may have reduced overhead for O3 code.

We found it necessary to implement the *profile record* stack using a statically allocated array for performance reasons. Implementing the stack using dynamic memory allocation (`malloc()` and `free()`) was too time-consuming for O3 code on some of our workload applications. For our benchmarks, we empirically found that a stack size of 1,000,000 entries was sufficient.

3.3 Trigger Mechanism

The original recompilation triggering mechanism in the ORP JVM involved decrementing the *method_entry* field in the target profile record from a value of 1000. Upon reaching 0, a recompilation was triggered and the O1 code of the method would be converted to O3 code. In our design, we re-used this mechanism by setting the *method_entry* field to a value of 1 when a recompilation was desired.

Our cycle counter-based compilation triggering decision was made at various locations. Upon detecting that the accumulated cycles for the target method was above a threshold and that a method had been entered at least 2 times, the *method_entry* field was set to a value of 1, forcing a recompilation upon the next invocation of the target method. This decision was evaluated in (1) the O1 start cycle function concerning the caller, (2) the O1 stop cycle function concerning the callee, (3) the O3 stop cycle function concerning the caller, and (4) in the `trigger_recompilation()` function concerning the caller who requested the recompilation of the callee method.

In our design, we decided that the recompilation time to convert an O1 method to O3 should be charged to no one. It did not appear appropriate to charge these cycles to the caller nor the target method. It's not the caller's fault that the target method required a recompilation. Nor should the target method be penalized for being recompiled to a higher optimization level. To implement this exemption, the `trigger_recompilation()` function pauses the caller and accumulates its *delta* cycles. It then sets the *m_last_cycle_count* field (which represents the start cycle value) to a value of 0 to signify to the callee to exempt charges to the caller method.

3.4 Further Details

Further implementation details can obtained from our source code. The source code to our modified version of the ORP JVM can be obtained from `http://www.eecg.toronto.edu/~tamda/ece1724/hwctr_orp.tar.gz`.

4. MEASUREMENTS AND RESULTS

4.1 Benchmarks

SPECjvm 98 was chosen as the benchmark package for this project. This specific benchmark package was chosen because the original ORP implementation used it.

It is necessary to point out that not all test cases in the SPECjvm 98 package were run, because a subset of them

use Java Native Invocations (JNI), which the ORP JVM does not support. Only those test cases listed in the JUDO paper [7] were attempted.

Compress, Jess, DB, Javac, Mpegaudio ran successfully without any problems. When the modified ORP was tested using Jack, the test case ran mostly problem-free, except the check-sum failed in the end. The modified ORP failed to run Mtrt.

4.2 Results

4.2.1 Baselines

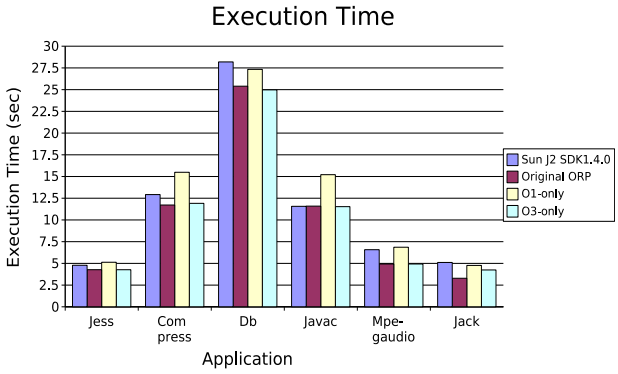

Figure 2: Execution time - baselines

In Figure 2, we compare the Sun JVM against the original ORP JVM to show some baseline numbers. The ORP JVM is run in 3 different modes. (1) Regular operation, which is byte-code to O1 to O3, (2) byte-code to O1 only, and (3) byte-code to O3 only. The first observation is that the standard ORP JVM configuration performs better than or equivalent to the Sun JVM on all workloads. The *O1-only* results verify that running just at O1 is suboptimal and that if you dynamically recompiled to O3 when appropriate, performance can be improved as shown in the *Original ORP* results. Surprisingly, the *O3-only* results indicate that compiling directly to O3 code is better than compiling to O1 code first, under all workloads. This result means that the O1 compiler is useless and that the O3 compiler should be used exclusively. What is also surprising is that the *O3-only* results indicate that it is very competitive with the *Original ORP* results, signifying that dynamic compilation was not very useful for these workloads on our particular hardware configuration. From these baseline results, our goal would be to attain execution times that are lower than the *Original ORP* and *O3-only* results.

4.2.2 Overheads

In Figure 3 we investigate overhead of our mechanisms. The *O1 call overhead* results indicate the overhead to O1 code of executing the extra inserted call function that calls the O1 start/stop functions. The O1 start/stop functions were nullified and contained only a `return;` statement. The O3 code did not contain our inserted code. The results show minimal overhead to O1 code. The *O1 no-trig* results indicate the overhead to O1 code of executing most of the O1 start/stop functions. The only action missing was the triggering mechanism. Therefore, the default triggering mechanism of 1000 method entries is in effect. Again, these results

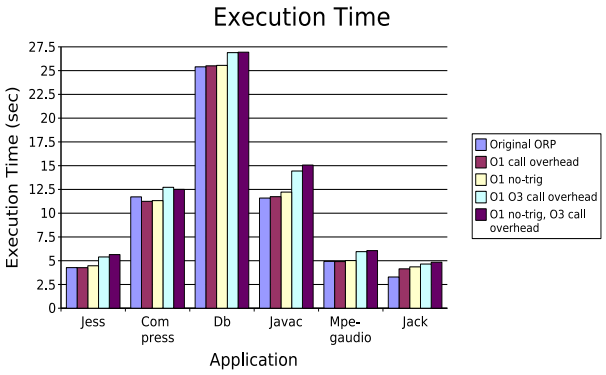

Figure 3: Execution time - overheads

show minimal overhead. The *O1 O3 call overhead* results indicate the overhead to both O1 and O3 code of executing the extra inserted call function that calls the O1 or O3 start/stop functions. The O1 and O3 start/stop functions were nullified and contained only a `return;` statement. The results indicate that for some applications there is noticeable overhead. O3 code is highly optimized and diverting it to perform housekeeping work for our enhancements is detrimental. For completeness, we show the *O1 no-trig, O3 call overhead* results. In this configuration, the O1 start/stop functions perform all of their duties except for the triggering mechanism, while the O3 start/stop functions are empty. The results verify again that the O1 enhancements add minimal overhead.

4.2.3 O3 Overheads

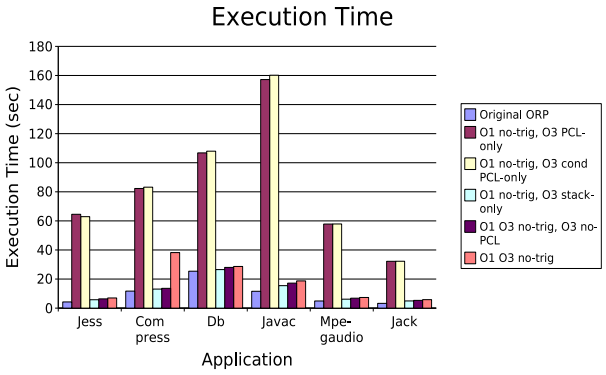

Figure 4: Execution time - O3 overheads

In Figure 4 we continue our investigation of the impact of our enhancements to O3 code. In these set of results, the O1 start/stop functions contain all except for the trigger mechanism. In the *O1 no-trig, O3 PCL-only* results, we take the empty O3 start/stop functions and add a call to the PCL library to read the current cycle count. This configuration measures the impact of the PCL library on O3 code. The results clearly indicate that there is an unacceptable amount of overhead. In the *O1 no-trig, O3 cond PCL-only* results, we call the PCL library only if the caller method is O1 code. This more accurately reflects the frequency of calling the PCL library since this work is not necessary if the caller is O3

code. Unfortunately, the results clearly show that overhead is extremely high. These results indicate that the use of *design decision B* was a bad choice. In contrast, *design design A* would not interfere with O3 code at all and by-pass the problem completely. As mentioned, due to the time-constraints, *design decision A* could not be implemented. To investigate the impact of accessing the *profile record* stack in O3 code (as required by *design decision B*, results are shown by the *O1 no-trig, O3 stack-only* bars. These results indicate that accessing the *profile record* stack is acceptable in O3 code.

To investigate the impact to O3 code with 99 % of our enhancements, we enabled all of the O3 start/stop function code except for the PCL library calls and the triggering mechanism, since the PCL calls appeared to have unacceptable overhead. The results, shown by the *O1 O3 no-trig, O3 no-PCL* bars, indicate acceptable overhead. Finally, the *O1 O3 no-trig* results show the impact of enabling all O3 start/stop function code except for the trigger mechanism. PCL library calls were made when necessary. To our confusion, the results show very acceptable overhead, contradicting the results from the *O1 no-trig, O3 cond PCL-only* bars. We re-ran the experiments several times in disbelief and obtained the same results. We have yet to determine the root cause of these inconsistencies. Fortunately, these results indicate that our mechanism, when mostly enabled (except for the trigger mechanism), has acceptable overhead.

4.2.4 Results Without Backup Triggers

In Figures 5 and 6, we measure the execution time of the SPECjvm 98 applications with our enhancements completely enabled. That is, the trigger mechanism is enabled, in contrast to the previous overhead investigation experiments. In Figure 5, we show the execution times in comparison to the original ORP JVM, while in Figure 6 we show the number of recompilations from O1 to O3 code. In this configuration, the original triggering mechanism of 1000 method entries and 10,000 taken loops was disabled. The JVM recompilation decision rested solely on our mechanism. We varied the recompilation threshold from 1,000 cycles to 10 billion cycles. We include an extra result, where we examine the results with the 10,000 taken loops trigger mechanism enabled.

Unfortunately, the results show that we could not perform better than the default ORP JVM. It is interesting to note that in all except for the *compress* workload, the recompilation threshold should be set below 100 million cycles. This result makes sense as follows. For long-running workloads, if a method spends more than 0.1 seconds in a method, then that method should be compiled. On our approximately 1.5 GHz processor, this would translate to roughly 150 million cycles. Unfortunately, the results indicate that the results are fairly insensitive once the threshold is set below 100 million cycles.

Despite this small aspect making sense, there is still a very large contradiction to the *O1-only* results shown in Figure 2. As the cycle count threshold is increased to infinity, the system should behave similar to the *O1-only* results. In those results, performance is not extremely bad. However, in the results in Figure 5, we see extremely large execution times when the compilation threshold is greater than 1 billion cycles. Due to time constraints, we have yet to explore the root cause of these inconsistencies.

For the *compress* workload, the *10k-loop, 1M-cycles* configuration performed significantly better than other configurations. The use of the taken-loops trigger mechanism was necessary for this workload.

Figure 6 illustrates the number of method recompilations from O1 code to O3 code. As the recompilation threshold is decreased, we see an expected increase in the number of methods recompiled. When trying to correlate this extra work to the execution times in Figure 5, there is a *disconnect*. The extra recompilation work results in no execution time improvements in the spectrum of 10 million cycles to 1,000 cycles.

4.2.5 Results With Backup Triggers

In Figures 7 and 8, we investigate execution time as in Section 4.2.4 but with the default backup trigger mechanism enabled. That is, if a method is entered more than 1000 times, or more than 10,000 taken-loops are seen, then the method is recompiled from O1 to O3. In addition to these default mechanisms, we add our cycle count trigger mechanism to compliment the system. In this way, the system will trigger recompilations no later than the default ORP JVM. This is in contrast to our configuration in Section 4.2.4 where recompilation may never occur.

Unfortunately, these results indicate that performance cannot be improved beyond the original ORP JVM configuration. The results seem insensitive to the cycle count recompilation threshold since there is relatively no change in execution times when exploring the full range of cycles.

The number of recompilations appears to make sense. As the cycle count threshold is decreased, the system performs more recompilations than the default ORP JVM. When the cycle count threshold is increased beyond a certain value, this triggering mechanism is effectively disabled and so the number of recompilations is equivalent to the original ORP JVM.

5. CONCLUSION AND FUTURE WORK

This paper explored a way of exploiting performance data stored in hardware counters on dynamic runtime systems.

Hardware counters offer a low overhead mechanism of obtaining valuable and precise runtime micro-architectural performance information that is otherwise unobtainable from software techniques alone. Many of these parameters can potentially offer accurate real-time performance information for the purpose of run-time instrumentation and improving triggering techniques for run-time code recompilation.

Only one hardware counter metric, the CPU Cycle counts, was tried in this project. Although the performance results were not as good as originally anticipated, the work done for this project has created an public infrastructure[1] for ex-

[1]The source code is open and can be obtained from `http://www.eecg.toronto.edu/~tamda/ece1724/hwctr_`

perimenting with the idea of using hardware counter data in general. The infrastructure created can easily be re-used to explore other hardware counter metrics available through the PCL package to create improved triggering mechanisms for dynamic run-time code recompilation.

Due to time constraints, a few originally planned features had to be *de-scoped*. The modified ORP currently only supports single-threaded programs, as multi-threading support was not implemented. This could be implemented in the future. Also, accounting for the garbage collection mechanism used in the original ORP was not included in the scope of this project. However, profiling using hardware counters that takes garbage collection into account can be easily implemented by stopping the counter when the garbage collection thread starts executing, and resuming the counter when the garbage collection thread stops, as described in Section 3.2.

Other than the *de-scoped* items listed above, some improvements can be made. The PCL overhead was incorporated into the cycle counts rather than being isolated. An initial test run of a simple `Hello World` program indicated that there was only around 10% overhead resulting from PCL. For a large program, the overhead cycles consumed by the PCL may be much smaller relative to the total number of cycles consumed by the entire program. It would still be possible to incorporate some tuning code to isolate and remove the PCL overhead when counting the number of cycles consumed by a method. Also, in this project the cycle count values obtained from hardware counters were used as replacement of the primitive counters in the original ORP implementation. It is possible to explore other options, as indicated in the Table 1 to see if there are better metrics obtainable from the hardware counters to trigger dynamic run-time code recompilation.

6. REFERENCES

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P.F. Sweeney. Adaptive optimization in the Jalapeno JVM. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA'00)*, October 2000.

[2] Matthew Arnold. Online instrumentation and feedback-directed optimization of Java. Technical Report Technical Report DCS-TR-469, Department of Computer Science, Rutgers University, 2002.

[3] Matthew Arnold and Barbara Ryder. A framework for reducing the cost of instrumented code. In *Programming Language Design and Implementation*, June 2001.

[4] V. Bala, E. Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Programming Language Design and Implementation*, June 2000.

[5] Rudolf Berrendorf and Bernd Mohr. PCL - the performance counter library: A common interface to access hardware performance counters on microprocesors. Technical Report Version 2.2, University of Applied Sciences Bonn-Rhein-Sieg and Research Centre Juelich GmbH Central Institute for Applied Mathematics, http://www.fz-juelich.de/zam/PCL/doc/pcl/pcl.pdf, 2002.

[6] Howard Chen, Wei-Chung Hsu, Jiwei Lu, Pen-Chung Yew, and Dong-Yuan Chen. Dynamic trace selection using performance monitoring hardware sampling. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, 2003.

[7] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Programming Language Design and Implementation*, June 2000.

[8] Michal Cierniak, Marsha Eng, Neal Glew, Brian Lewis, and James Stichnoth. The Open Runtime Platform: A flexible high-performance managed runtime environment. *Intel Technology Journal*, 7(1):5–18, February 2003.

[9] Thomas Kistler and Michael Franz. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems*, 25(4):500–548, July 2003.

[10] Chandra Krintz and Brad Calder. Using annotations to reduce dynamic optimization time. In *Programming Language Design and Implementation*, June 2001.

`orp.tar.gz`.

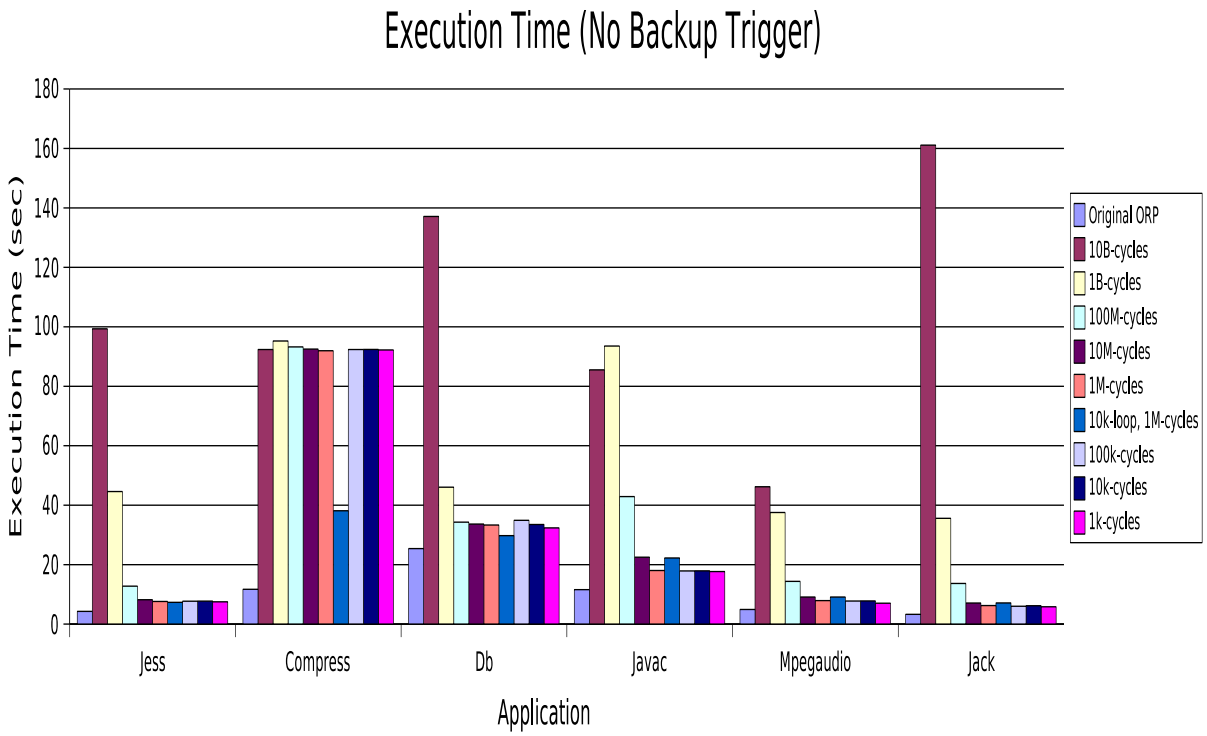

Figure 5: Execution time - no backup trigger

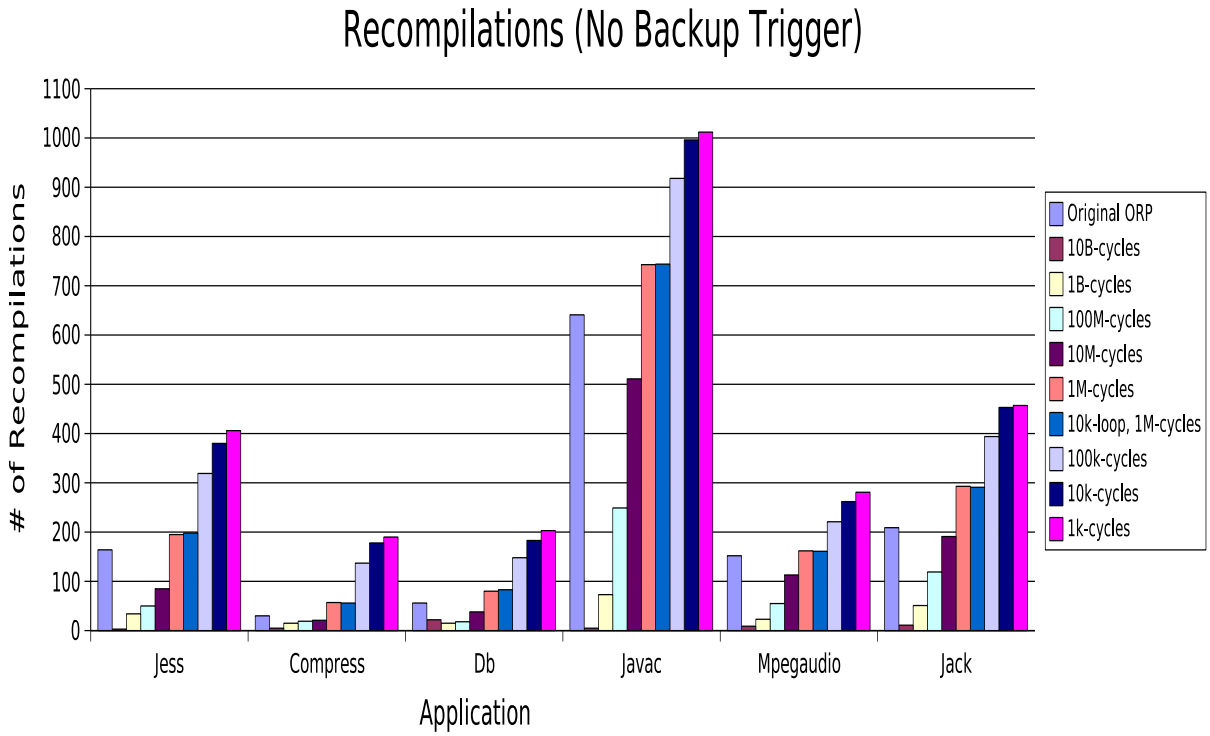

Figure 6: Recompilations - no backup trigger

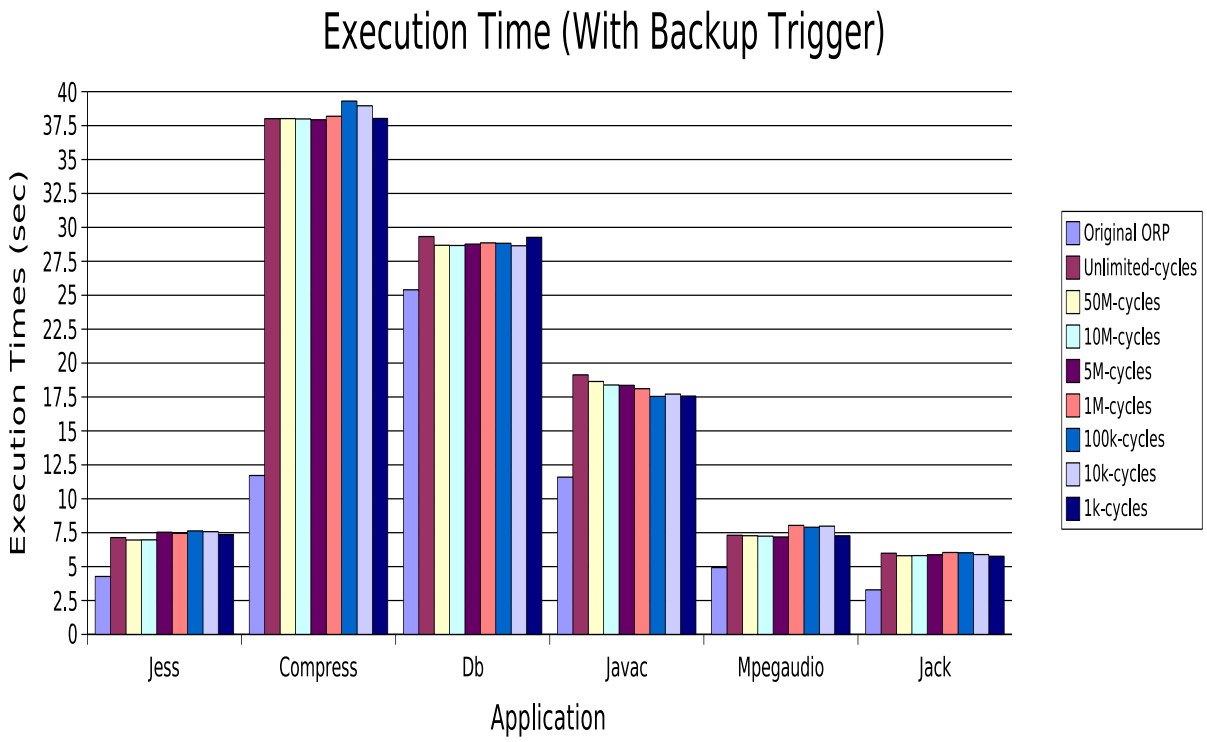

Figure 7: Execution time - with backup trigger

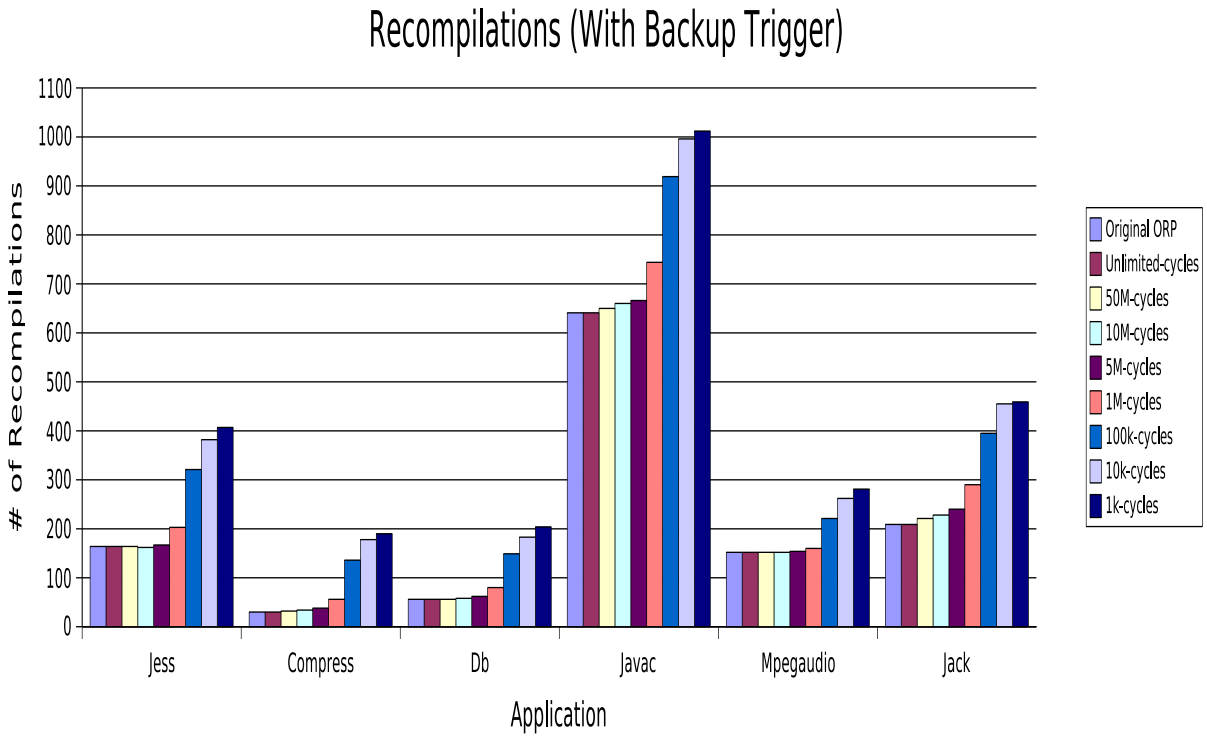

Figure 8: Recompilations - with backup trigger