Improving Performance on Incremental Compilation of Java Bytecodes

Tarique Kazi

Follow this and additional works at: https://louis.uah.edu/honors-capstones

Part of the Programming Languages and Compilers Commons

Recommended Citation
https://louis.uah.edu/honors-capstones/435

This Thesis is brought to you for free and open access by the Honors College at LOUIS. It has been accepted for inclusion in Honors Capstone Projects and Theses by an authorized administrator of LOUIS.
Abstract

To obtain good performance for Java program distributed as Java Virtual Machine (JVM) bytecode, a translator is used to convert the bytecodes into native machine language as needed. This lazy translation method, Just-In-Time (JIT) compilation (also known as incremental compilation), delays translation of a method until the method is called. This intersperses execution with translation, allowing the users to see some progress being made by the program rather than waiting for the entire program to be compiled before any instruction is executed, and avoids translations of unused methods. However, in some cases JIT translation can take a significant amount of time to perform.

A group of representative Java benchmarks was run using instrumented Kaffe JVM v0.10 to find out the time spent in JIT and native execution. Using the results, the possibility of overlapping JIT and native execution was examined. This also helped in calculating the possible performance improvement (speed up) gained by the overlapping strategy without modifying the application code.

1 Introduction

The Java bytecode language [8] is emerging as a standard for software distribution. The machine-independent Java bytecode programs are expected to run without modification on multiple platforms. Such run-time environments depend on a Java Virtual Machine (JVM) program that interprets the Java bytecode directly for execution. However, this interpretation takes a significant amount of time for each instruction and thus reduces the performance as compared with the native-code execution. The Just-In-Time (JIT) translator, on the other hand, translates Java bytecode into native machine language on demand. With this translator, interpretation of individual JVM bytecodes never takes place. Therefore, JIT approach avoids the performance penalty of interpretation. It also avoids compilation of Java VM code
units that are never executed, thus the overhead of compiling unused code is avoided. However, JIT uses a single thread of control that alternates between compilation into native machine language and execution of the compiled object code [10]. Because compilation is interspersed with execution, the system would appear to pause whenever compilation occurs interrupting the execution. A significant disadvantage of JIT is that it does not save the native code sequences in external files for future invocations of the same program. Rather, JIT compilers generate on-demand code of executed Java methods everytime the program is started and cache the native code sequences to speed up the processing of the execution of methods for future calls [6]. This necessity of explicitly generating code and caching native code, in some cases, requires more effort for the JIT than interpreter. Because of the translation overhead, JIT may use more time than saved by native code execution. For such cases, direct interpretation of JVM bytecodes would be faster to finish the computation.

This paper investigates the possible performance improvement (speed up) from overlapping the compile time and execution time for Java applications without modifying the code. MultiKron II [9] instrumented Kaffe JVM v0.10 was used to run a group of representative Java Benchmarks and the results were studied. Ideally, a speed up of two would be observed if the compile time and execution time for individual code segments overlap exactly. The results of this experiment show that the overlapping strategy, without modifying the application code, works better for Java applications with short runtimes. The Java applications with runtime of less than 5 seconds achieved a significantly higher speed up (6% - 9%) compared to the applications with longer runtime (2% or less speed up). Section 2 describes the other researchers' work in Java performance and JIT analysis. Section 3 discusses the benchmarks used. Section 4 describes the Kaffe JVM with MultiKron II instrumentation board, and the method of collecting data. Section 5 describes the experiments and the collected data, a discussion of the data follows in section 6. The plans for future work are in section 7 and section 8 summarizes the results.

2 Related Work

Even though other researchers have investigated compilation models for reduced compilation costs and improved runtime performance, similar investigations for Java distributed applications using Java benchmarks have not been performed yet. Plezbert and Cytron [10] suggested a continuous compilation model for dual-processor systems where JIT can overlap with interpretation/execution for better performance. This work is very close to the work described in this paper. However, Plezbert and Cytron used some selected C benchmarks (since Java applications were
unavailable at the time) to predict program behavior of web distributed applications. The work described in this paper not only used actual Java applications for investigation but also used instrumented JVM with high resolution MultiKron II board [9] for accurate data collection (time spent in compiling units and time spent in executing units). As a result, more performance estimates were obtained for the selected Java applications.

Similarly Hsieh and et. al. [6] discussed their initial prototyping investigation with a compiler called NET (Native Executable Translator) that outperformed MS-JIT or Sun Interpreter at the time, but the authors used non-Java benchmarks (C/C++ benchmarks) for their work as well. The problem with C/C++ benchmarks is that they are distributed in a machine-dependent form unlike Java applications. Thus, their results are not applicable to distributed Java code such as applets or code that makes use of Remote Method Invocation (RMI). The approach described in this paper will yield a performance improvement for applications that require the program to be distributed as Java bytecodes.

3 Benchmarks

This study was conducted on a set of Java benchmark applications developed by others. The five benchmarks used are Linpack, JavaCup, JLex, CaffeineMark, and VolanoMark. They were chosen for this experiment because of their variety in sizes, runtimes, and natures. Each benchmark is concisely described in the following subsections:

3.1 Linpack

The Linpack [3] benchmark is a numerically intensive test that has been used for years to measure the floating point performance of computers. However, this test of Java version is more a reflection of the state of the Java systems than of the floating point performance of the underlying processors. The code implements a solution of a dense 100x100 system of linear equations with one right hand side, Ax=b. The matrix is generated randomly and the right hand side is constructed so the solution has all components equal to one. The method of solution is based on Gaussian elimination with partial pivoting.

3.2 JavaCup v0.10g

Java Based Constructor of Useful Parsers (JavaCUP) [7] is a system for generating LALR parsers from simple specifications. It serves the same role as the widely used program YACC and offers most of the features of YACC.
However, CUP is written in and operates entirely with Java code, uses specifications including embedded Java code, and produces parsers which are implemented in Java.

### 3.3 JLex (JavaLex) v1.2.3

JLex [1] is a lexical analyzer generator, written for Java, in Java. A lexical analyzer groups characters in an input stream into tokens. The JLex utility is based upon the Lex lexical analyzer generator model. Lex is a lexical analyzer generator for the UNIX operating system, targeted to the C programming language. Lex takes a specially-formatted specification file containing the details of a lexical analyzer. This tool then creates a C source file for the associated table-driven lexer. JLex takes a specification file similar to that accepted by Lex, then creates a Java source file for the corresponding lexical analyzer.

### 3.4 CaffeineMark 3.0

The CaffeineMark [12] is a Java applet that measures the performance of a system. The CaffeineMark 3.0 is a series of tests that measure the speed of Java programs running in various hardware and software configurations. CaffeineMark scores roughly correlate with the number of Java instructions executed per second, and do not depend significantly on the amount of memory in the system or on the speed of a computer's disk drives or internet connection. The CaffeineMark uses 9 tests to measure various aspects of JVM performance. Each test runs for approximately same length of time. The score for each test is proportional to the number of times the test was executed divided by the time taken to execute the test. The following is a brief description of what each test does:

- **Sieve**: The classic sieve of eratosthenes finds prime numbers.
- **Loop**: The loop test uses sorting and sequence generation as to measure compiler optimization of loops.
- **Logic**: Tests the speed with which the virtual machine executes decision-making instructions.
- **String**: Tests the speed of JVM string handling functions.
- **Method**: The Method test executes recursive function calls to see how well the VM handles method calls.
- **Float**: Simulates a 3D rotation of objects around a point.
- **Graphics**: Draws random rectangles and lines.
- **Image**: Draws a sequence of three graphics repeatedly.
- **Dialog**: Writes a set of values into labels and edit boxes on a form.
The overall CaffeineMark score is the geometric mean of the individual scores, i.e., it is the 9th root of the product of all the scores.

3.5 VolanoMark v1.0

VolanoMark [13], a server-side Java benchmarking tool for assessing the performance and stability of any JVM. Java server applications for multi-user environments require a different kind of benchmark than CaffeineMark to assess JVM performance accurately. Multi-user Java servers tend to be highly multi-threaded, with at least one thread per connection, and highly networked, with a large number of long lasting socket connections. In addition, these applications must be highly stable and highly scalable, capable of supporting hundreds or thousands of concurrent client connections without any degradation in response time. Using Volano's Java-based communications engine, VolanoMark creates ten groups of 20 connections each for a total of 200 connections with the server. Each client connection takes turns broadcasting 10 messages to its respective group. Upon completion, VolanoMark reports the score as the average number of messages transferred per second.

4 Methodology

One of the objectives was to find out the time spent in JIT and native execution so that the performance improvement could be calculated for overlapping JIT and native execution without modifying the application code. The host machine consisted of a 100MHz Pentium with 96MB of memory running the Redhat 4.2 distribution of Linux. The National Institute of Standards and Technologies (NIST) MultiKron II board [9] with a PCI interface provides a high-resolution time base (100ns). The board time-stamps each data write and can record approximately 880,000 events in the 16MB of local memory. The timing data can be downloaded from the board upon completion of the program execution. The host machine was equipped with a MultiKron II board to collect timing information for JIT and native execution. A software utility was developed in order to analyze the collected timing information. Kaffe Java Virtual Machine v 0.10 [11] was used as the runtime environment.

Modifications were made to the Kaffe JVM for this project. Instrumentation instructions were inserted at proper places of the JVM code so that the beginning and the end of various events (JIT, thread operation, method call, and garbage collection) were logged to the MultiKron II board. The Kaffe JVM initializes the board before a Java program starts execution. During the execution of the Java program, events are recorded to the MultiKron II board.
After the completion of execution, the collected data is read from the MultiKron II board using a MultiKron utility program, and stored in a data file. A software utility was developed to analyze the data from the data file. Individual JIT times were calculated by subtracting a JIT-start-time from its JIT-stop-time. The time segments spent in execution were calculated by subtracting a JIT-stop-time from the following JIT-start-time. This way the program runtime was divided into alternating segments of JIT compilation of bytecode and execution of translated methods. The total time spent in JIT and total time spent in execution were calculated. The summation of total JIT and total execution gave the total runtime for an application. To analyze the benefit of overlapping JIT and execution, each JIT time segment was compared with the previous execution time segment and the longer of those two time segments were counted as the runtime for that overlapping pair. All of these overlapping runtimes were added together to get the total runtime for the same application with overlapping. The ratio of the original runtime and the overlapping runtime was calculated to see the performance improvement gained for this application.

5 Experiments

The five major benchmarks used in this experiment to collect data (Linpack, CaffeineMark, VolanoMark, JavaCup, and JLex) have been compiled and run successfully. Then the results were collected and analyzed using the methodology described in section 3. This particular methodology calculates the speed up for worst case scenarios because the application code was never modified and while overlapping JIT segments with corresponding previous execution segments, always the longest of the pair was counted as runtime. While analyzing the collected data, it was noticed that for every benchmark JIT spent around 1.73 seconds compiling one huge Java method (java/lang/Character.<clinit>). This large amount of time that JIT spent for just that one particular method was very unusual compared to all other methods because this character class initializer method sets values for each element in a very large array every time a program is run and the initializer has 4 bytecode instructions for each element in that array, yielding an immense method. This single method (let us call it “character class”) significantly affected the overall performance every single time it was compiled by JIT translator. In order to maintain consistency throughout the compilation and execution of Java methods, the final results were calculated factoring out the “java/lang/Character.<clinit>” method from the calculation. The table below summarizes the initial performance improvement (with the “java/lang/Character.<clinit>”) and the final performance improvement (without the “java/lang/Character.<clinit>”) for each benchmark as calculated by the methodology described in section 3:
Table 1: Performance gained by overlapping JIT and native execution for Java applications without modifying the code

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Runtime (sec.)</th>
<th>Initial Performance Improvement</th>
<th>Final Performance Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linpack</td>
<td>3.97</td>
<td>2.60%</td>
<td>6.0%</td>
</tr>
<tr>
<td>JavaCup</td>
<td>4.81</td>
<td>5.81%</td>
<td>8.98%</td>
</tr>
<tr>
<td>JLex</td>
<td>17.9</td>
<td>1.01%</td>
<td>1.11%</td>
</tr>
<tr>
<td>CaffeineMark</td>
<td>28.9</td>
<td>0.33%</td>
<td>0.35%</td>
</tr>
<tr>
<td>VolanoMark</td>
<td>300</td>
<td>0.23%</td>
<td>0.24%</td>
</tr>
</tbody>
</table>

6 Discussion of Results

The information produced by the software utility helped in producing linear graphs (Figure 1 through 10) for pictorial analysis. For each of the benchmarks two graphs were produced. First graph shows the data points for JIT/wallclock vs. wallclock, and the second graph shows the data points for execution (i) vs. JIT (i+1). First graph shows the JIT behavior over the runtime. Downward slope in the curve represents execution mode and upward slope in the curve represents compilation mode. With exact overlapping of JIT compilation and execution, ideally the data points should cluster across a horizontal line over the runtime intersecting the y-axis at 0.5. The second graph shows the overlapping feasibility of execution and the corresponding following JIT. Ideally data points should cluster around y=x line. A trendline is derived from the generated data points to visually compare with y=x line. The graphs are as follows:

**Linpack**

![JIT/Wallclock vs. Wallclock for Linpack](image-url)
Figure 2: native execution (i) vs. JIT (i+1) for Linpack

JavaCup

Figure 3: JIT behavior for JavaCup
Figure 4: native execution (i) vs. JIT (i+1) for JavaCup

Figure 5: JIT behavior for JLex
Figure 6: native execution (i) vs. JIT (i+1) for Jlex

CaffeineMark

Figure 7: JIT behavior for CaffeineMark
Execution (i) vs. JIT (i+1) for Caffeine

**Figure 8:** native execution (i) vs. JIT (i+1) for CaffeineMark

VolanoMark

**Figure 9:** JIT behavior for VolanoMark
If we study the graphs of JIT behavior (Figure 1, 3, 5, 7, and 9), we notice that JIT compilation for Java methods take longer time at the beginning of the program execution and less time is spent in native execution. As the program execution progresses, compilation time keeps decreasing and more time are dedicated for execution. Throughout the curve we see a general tendency of downward slope which tells us that as more and more time are elapsed, JIT events are less frequent and more native code execution is taking place than at the early stage of execution. The only noticeable upward spike (around 1.73 seconds) in the curve indicates the “java/lang/Character.<clinit>” method translation (time spent compiling the character class initializer). This overall behavior was expected. At the beginning of execution there are very little native code available for execution, so the JIT translator spends more time in compiling Java methods. As the execution progresses, more and more Java methods are already translated into native codes and thus less remaining to be translated. So JVM spends more time executing and less time translating (short compile duration) the code. This trend actually poses a problem for applications with long runtime (runtime longer than 5 seconds). If we carefully study the graphs of JIT behavior, we see that the curves start going downward around the 3rd to the 5th seconds of the runtime. That means by the 3rd to the 5th second of program execution, JIT completes translating most of the bytecode segments and thus little compilations remain to be done compared to the execution. This imbalance in remaining JIT and execution effects the overlapping process. So the programs with runtime longer
than 5 seconds does not gain much speed up after the $5^{th}$ second even with overlapping. Maximum overlapping benefit is gained from the start to the $3^{rd}$ second of program execution for any application.

The second phase of the graphs (Figure 2, 4, 6, 8, and 10) show us a graphical picture of consecutive execution time vs. JIT time. This gives us an idea of the overlapping feasibility between execution time and the following translating time. Ideally, all points should cluster around the $y=x$ line. In reality, we see that most of the points are clustered below the $y=x$ line indicating that most of the JIT segments took more time to be translated than the execution time of the previous segments. This observation shows that there is certainly room for runtime performance improvement by reducing JIT overhead so that there is a more balanced overlapping between JIT segments and execution segments.

Table 1 shows the performance improvements gained for selected benchmarks by overlapping the consecutive native execution and JIT translation using two threads of control. The wide range in the performance improvements gained is because of the variety of sizes and codes of those benchmark applications. An ideal speed up would be a factor of 2 where each JIT translation is exactly overlapped by a native execution. That would require much more intensive research in order to improve the virtual machine environment, and possibly modifying the application code as well so that there is a proper balance in translating and executing Java methods.

7 Future Work

One of the main problems encountered in this work was to find large Java benchmarks that are developed in Java. Sources of large Java applications for testing purposes are still scarce. The extension of this research will obviously involve many more Java applications of various sizes and characteristics to test. The Java virtual machine environment could be improved so that it handles the "character class" differently and it would not affect the performance so badly. An improved JVM could also make use of idle processors in a multiprocessor system to look far ahead into the code and build a call graph of to-be-translated methods so that future code segments could be compiled ahead for execution. This would hide the translation overhead greatly and could provide better runtime performance. Using an actual multiprocessor system for this work would be helpful where separate processors can be dedicated for translation and native execution. A perfect speedup may not be attainable, but these suggested extensions on this work could significantly improve Java runtime performance.
8 Conclusion

While Java bytecode language has been accepted as the standard for machine-independent software distribution, parallel computing has provided us the means for higher performance. Machines with multiple processors are becoming affordable to the user community. This fact acted as the key motivation for this investigation so that parallel processing of Java applications can provide a means to reduce the JIT overhead yielding significant improvement on runtime performance. The results presented in this paper are truly exciting for Linpack and JavaCup indicating that great performance benefits could be obtained for Java applications with runtime of 5 seconds or less by overlapping JIT and execution. Larger applications (Jlex, CaffeineMark, and VolanoMark), even though failed to show promising improvement on performance from overlapping, may benefit from overlapping in the future with improved JVM and dedicated multiprocessors running the applications as suggested in section 7. The work described in this paper assumes a dual processor system; systems with more processors are anticipated to yield higher performance improvement.

9 Acknowledgements

I am grateful to my advisor Dr. William Cohen for his guidance in this work. I gratefully acknowledge the research grant and continuous support from the UAH Honors Program, Dr. Richard Modlin, and Betty Cole. I would also like to thank the National Institute of Standards and Technology for the loan of MultiKron instrumentation boards for this project.
References


/* utility.h - This is the header file of the utility program that analyzes */
/* the recorded data from the NIST MultiKron II board after the */
/* run of a Java benchmark application. */
/* */
/* SYSTEM: x86 Linux */
/* BY: Tarique H Kazi (based on Dr. William Cohen's Static) */
/* UAH */
/* */
/* HISTORY: */
/* 07/14/1998 Final version */
/* This is the header file for the utility program...... */
/* *************************************************************************/

#define TRUE 1
#define FALSE 0

#define PROC_BITS 4
#define MAX_PROC (1<<PROC_BITS) /* number of processors in system */

/* some defines used to unpack the lower n bits of user data which contains */
the process/processor number and event number as such:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>proc</td>
<td>op</td>
<td>event</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----------</td>
</tr>
</tbody>
</table>

#define PROC_ID_SIZE 4
#define OPERATION_SIZE 8
#define EVENT_DATA_SIZE 20
#define TID_SIZE EVENT_DATA_SIZE

#define PROC_ID_MASK (( ( 1 << PROC_ID_SIZE ) - 1) << 
(OPERATION_SIZE + EVENT_DATA_SIZE ))

#define GET_PROC_ID(x) ( (x) & PROC_ID_MASK) >> 
(OPERATION_SIZE + EVENT_DATA_SIZE )

#define OPERATION_MASK (( ( 1 << OPERATION_SIZE ) - 1) << 
(EVENT_DATA_SIZE )

#define GET_OPERATION(x) ( ((x) & OPERATION_MASK) >> EVENT_DATA_SIZE)

#define EVENT_MASK (( ( 1 << EVENT_DATA_SIZE ) - 1) )

#define GET_EVENT(x) ( ((x) & EVENT_MASK) )

/* JIT defines */
#define JIT_START_OP      (0x01)
#define M_JIT_START_OP    (0xff)
#define JIT_STOP_OP       (0x02)
#define M_JIT_STOP_OP     (0xff)
#define JVM_GC_START_OP   (0x21)
#define M_JVM_GC_START_OP (0xff)
#define JVM_GC_STOP_OP    (0x22)
#define M_JVM_GC_STOP_OP  (0xff)
struct event /* event record */
{
    int header; /* event number */
    double time_stamp;
    unsigned long source_id;
    unsigned long data_hi;
    unsigned long data_lo;
};

/* defines for reading in event log */
#define DATA_LENGTH 20
#define RESOURCE_SAMPLE_MASK 0x18000000
#define RESOURCE_SAMPLE_ 0x18000000
#define GET_HEADER(X) (((X) >> 24) & 0xff)
#define GET_TIMESTAMP(HI, LO) ((HI & 0x00ffffff) * 4294967296. + LO)

extern int main(int argc, char *argv[]);
extern int open_file(char *filename, int flags);
extern void process_file(int data_file);
extern int read_event(int fd, struct event *x);
extern void process_options(int argc, char *argv[]);
/* utility.c - This is the main program of the utility program that analyzes */
/* the recorded data from the NIST MultiKron II board after the */
/* run of a Java benchmark application. */
/* */
/* SYSTEM: x86 Linux */
/* Tarique H Kazi (based on Dr. William Cohen's Static) */
/* UAH */
/* */
/* HISTORY: */
/* 07/14/1998 Final version */
/* */
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <values.h>
#include "utility.h"

char program[] = "utility";
char version[] = "1.0";
char *file_name = "mk.dat";
int data_file;

/* Option variables */
int summarize_performance=TRUE;
int JIT_behavior=TRUE;
int summarize_overlap=TRUE;

/* Option List Stuff */
void p_summarize_performance() {summarize_performance = TRUE;}
void p_nsummarize_performance() {summarize_performance = FALSE;}
void p_JIT_behavior() {JIT_behavior = TRUE;}
void p_nJIT_behavior() {JIT_behavior = FALSE;}
void p_summarize_overlap() {summarize_overlap = TRUE;}
void p_nsummarize_overlap() {summarize_overlap = FALSE;}
void p_file(char *s) {file_name = s;}

typedef struct {
    char *option;
    int arg;
    void (*process)();
    char *descr;
} Opt;

Opt options[] = {
    {"+sp", 0, p_summarize_performance, "Performance summary" },
    {"-sp", 0, p_nsummarize_performance, "No performance summary" },
    {"+jb", 0, p_JIT_behavior, "Print JIT behavior data" },
    {"-jb", 0, p_nJIT_behavior, "Do not print JIT behavior data" },
    {"+so", 0, p_summarize_overlap, "Overlapping summary" },
    {"-so", 0, p_nsummarize_overlap, "No overlapping summary" },
    {"", 0, p_file, ""}, /* anything else is a file */
    {NULL, 0, NULL }
};
int main(int argc, char *argv[])  
{
    fprintf(stderr, "%s Version %s 1998\n", program[0], version[0]);
    if ( argc == 1 )
    {
        Opt *p = options;
        fprintf(stderr, "%s [options] f1 f2 ... fn\n", argv[0]);
        while ( *(p->option) != '+' )
        {
            fprintf(stderr, "\t%s %s\t%s\n", p->option, p->arg, p->descr);
            p++;
        }
    }else{
        ProcessArgs(argc-1, &argv[1], options);
        data_file = open_file(file_name, O_RDONLY);
        process_file(data_file);
    }
    return ;
}

int open_file(char *file_name, int flags)
{
    int fd;

    fd = open(file_name, flags);
    if ( fd == -1 )
    {
        fprintf(stderr,"Can't open %s\n",data_file), exit(-1);
        return fd;
    }

    void process_file(int data_file)
    {
        int tid; /* thread id */
        int samples = 0;
        struct event x;
        double last_time_stamp1=0.0;
        double last_time_stamp2=0.0;
        double last_time_stamp3=0.0;
        double last_time_stamp4=0.0;
        double time=0.0;
        double extime=0.0;
        double jittime=0.0;
        double startup=0.0;
        double sum=0.0;
        double wall_t=0.0;
        double jit_offset=0.0;
/* read in entries and process them */
while (read_event(data_file, &x)) {
    int operation;
    int proc;
    ++ samples;

    operation = GET_OPERATION(x.data_lo);
    tid = GET_EVENTTX.data_lo;
    proc = GET_PROC_ID(x.data_lo);

    if ((operation & M_JIT_START_OP) == JIT_START_OP) {
        last_time_stamp1=x.time_stamp;
        last_time_stamp3=(last_time_stamp1-last_time_stamp2)/(10000.0);
        if ((tid-1)==0)
            startup=last_time_stamp3;
        else {
            if(summarize_performance) {
                fprintf(stderr, "Execution %d time = %f msec\n", tid-1, last_time_stamp3);
            }
            extime=extime+last_time_stamp3;
        }
    }
    else if ((operation & M_JIT_STOP_OP) == JIT_STOP_OP) {
        last_time_stamp2=x.time_stamp;
        last_time_stamp4=(last_time_stamp2-last_time_stamp1)/(10000.0);
        if (tid==1){
            last_time_stamp4=last_time_stamp4+startup;
            jit_offset=last_time_stamp4;
            if(summarize_performance) {
                fprintf(stderr, "JIT %d time = %f msec\n", tid, last_time_stamp4);
            }
        }
        else if(summarize_performance) {
            fprintf(stderr, "JIT %d time = %f msec\n", tid, last_time_stamp4);
        }
        if (last_time_stamp4 > last_time_stamp3)
            sum=sum+last_time_stamp4;
        else
            sum=sum+last_time_stamp3;
        jittime=jittime+last_time_stamp4;
        if(tid==1)
            wall_t=jit_offset;
        else
            wall_t=wall_t+last_time_stamp3+last_time_stamp4;
        if(JIT_behavior) {
            fprintf(stderr, "%f, %f\n", wall_t, jittime/wall_t);
        }
if(summarize_overlap) {
    fprintf(stderr, "%f, %f\n", last_time_stamp4,
last_time_stamp3);
}
}

if(summarize_performance) {
    fprintf(stderr, "\n\nTotal JIT compilation time = %f msec\n",
jittime);
    fprintf(stderr, "Total execution time = %f msec\n", extime);
    fprintf(stderr, "Total time for a single processor = %f msec\n",
jittime+extime);
    fprintf(stderr, "Sum of (Max(JIT, prev_execution)) = %f msec\n",
sum);
    fprintf(stderr, "\nPerformance gain (for dual-processor systems) = %f\n\n", (jittime+extime)/sum);}

/* attempt to read a event from the log file */
/* returns number of bytes read (0 implies end of file) */
int read_event(int fd, struct event *x) {
    unsigned long input[5], input[16];
    int bytes_read, temp;

    /* read in raw data sample*/
    bytes_read = read(fd, input, DATA_LENGTH);
    if (temp == -1)
        perror("error reading input data"), exit(-1);

    /* if this is a resource sample, read the rest of the sample
    (the resource counters) and process the data the same way as
    a trace sample */
    if ((input[0] & RESOURCE_SAMPLE_MASK) == RESOURCE_SAMPLE) {
        if ((temp = read(fd, rsrc, sizeof(rsrc))) < 0) {
            perror("error reading past resource counters");
            exit(5);
        }
        bytes_read += temp;
    }
    /* package event in structure */
    x->header = GET_HEADER(input[0]);
    x->time_stamp = GET_TIMESTAMP(input[0], input[1]);
    x->source_id = input[2];
    x->data_hi = input[3];
    x->data_lo = input[4];

    return bytes_read;
}
ProcessArgs(argc, argv, options)
int argc;
char **argv;
Opt *options;
{
    Opt *p;

    while ( argc-- > 0 )
    {
        p = options;
        while ( p->option != NULL )
        {
            if ( strcmp(p->option, "*")) == 0 ||
                strcmp(p->option, *argv) == 0 )
            {
                if ( p->arg )
                {
                    (*p->process)( *argv, *(argv+1));
                    argv++;
                    argc--;
                }
                else
                    (*p->process)( *argv );
                break;
            }
            p++;
        }
        argv++;
    }
makefile - Page 1 of 1

# make file for utility program that generates information about
# JIT performance

CFLAGS = -g

utility : utility.o

utility.o : utility.c utility.h
1. If you just type "utility" in the command line, you will see something like this:

utility Version 1.0 1998
utility [options] f1 f2 ... fn
   +sp Performance summary
   -sp No performance summary
   +jb Print JIT behavior data
   -jb Do not print JIT behavior data
   +so Overlapping summary
   -so No overlapping summary

2. If you just want to see the performance summary, then type as following:

utility +sp -jb -so

3. If you just want to see the overlapping summary, then type the following:

utility -sp -jb +so