RunAssert: A Non-Intrusive Run-Time Assertion for Parallel Programs Debugging

Chi-Neng Wen, Shu-Hsuan Chou, Tien-Fu Chen
Computer Science and Information Engineering
National Chung Cheng University
ChiaYi, Taiwan
{dave.tw, csh93chou, tfuchen}@gmail.com

Tay-Jyi Lin
SoC Technology Center,
Industrial Technology Research Institute
HsinChu, Taiwan
tjlin@twins.ee.nctu.edu.tw

Abstract—Multicore environments are rapidly emerging and are widely used in SoC, but accompanying parallelism programming and debugging impact the ordinary sequential world. Unfortunately, according to Heisenberg's uncertainty principle, the instrument trying to probe the target will cause probe effects. Therefore, current intrusive debugging methodologies for sequential programs cannot be used directly in parallel programs in a multicore environment. This work developed a non-intrusive run-time assertion (RunAssert) for parallel program development based on a novel non-uniform debugging architecture. Our approaches are as follows: (a) a current language extension for parallel program debugging (b) corresponding non-intrusive hardware configuration logic and checking methodologies and (c) several reality cases using the extensions mentioned above. In general, the target program can be executed at its original speed without altering the parallel sequences, thereby eliminating the possibility of probe effect. The net hardware cost is relatively low, the reconfigurable logic for RunAssert is 0.6%-2.5% in a NUDA cluster with 8 cores, such that RunAssert can readily scale up for increasingly complex multicore systems.

I. INTRODUCTION

Heterogeneous multicore environments are becoming more commonly used in current SoC. In general, a heterogeneous multicore architecture is composed of a general purpose central process unit (CPU), special purpose processors (DSP), and several application specific integrated circuit (ASIC) accelerators. In order to achieve parallelism when programming, many programming models with shared memory architecture have been proposed. Those programming models typically are categorized into two major branches, control parallelism and data parallelism. Control parallelism makes code or function parallelization in the program. In addition, control parallelism must follow the behavior of sequential consistency (SC) [1]. In order to maintain SC, programming models use many techniques to synchronize the parallelization codes. It is very easy to break SC in a parallel program, and the evidence of the break will not be easy to be found. However, traditional profiling and debugging jobs may intrusively probe to the target program and that may cause another SC problem. We proposed a toolset called RunAssert that seamlessly introduces an idea into the debugging operations and ordinary codes. This toolset can take the target program be assert during runtime without break the SC property, in other words, this is a non-intrusive debugging facility.

Contributions. The key idea of this work is that we have coupled the debugging information into the target program for readability and maintenance and then decoupled execution during runtime. This work is based on a flexible and lightweight multicore debugging architecture, a non-uniform debugging architecture (NUDA) [2], which guarantees that any operations declared by RunAssert will not interrupt the ordinary program. The debugging activity and ordinary program do not influence each other. Only when debugging events occur, reaching a breakpoint (the true condition of a watchpoint), will NUDA synchronously stop all of the cores and then switch the system control into a third party debugger through a in-circuit emulator (ICE).

There are two major contributions in this work:

1. A new concept for parallel program debugging: we think that debugging information should remain in the source code and be easily controlled by an on/off switch. Debugging information should not decrease the normal program execution speed.

2. A parallel program programming model for debugging: we used directive #pragma in C language as an example to show the programming model for RunAssert. This programming model is composed of directives, RunAssert functions and assertion expressions. Users can specify the assertion type, range, scope and target of directives and then define particular rules by RunAssert functions. Finally, assertion expression is an option used to indicate the particular conditions of the debugging rules.

II. PRELIMINARIES AND RELATED WORKS

The focus of this paper is using hardware to support debugging language to help programmers attack bugs in parallel programs. Related works on parallel program debugging can be divided into solutions based on software and solutions based on hardware. Some software-related solutions, such as Eraser [3], which uses intrusive software instruments, probe all of the lock/unlock operations and memory access events, such that data races are detected according to the analysis of those traces. However, intrusive instrumentation
changes the execution sequence and also contributes to the probe effect. It may also report many potential false positives, which can be prevented if a previous relation is considered for those shared accesses [4]. Consequently, those tools [5] generate impractically long traces and unacceptably imprecise results.

HARD [6] is a hardware approach that adds some additional fields to the cache for run-time race detection. The OCP-IP [7] debugging group published the standard debugging interface socket for multicore interconnection, which provides a non-intrusive debugging channel through which NUDA will communicate with debug co-processor in core. Hence, RunAssert will not be influenced by any Heisenberg effect.

### III. Runtime Assertion

This work developed a non-intrusive parallel program debugging/guarding mechanism that helps users to identify and locate the bugs hidden in parallel programs. Meanwhile, this work also provides on the fly assertion and exception handling even if the assertion is no longer in use.

#### A. Programming Model for Non-Intrusive Parallel Program Debugging

The concept of the non-intrusive programming model for parallel program debugging is decoupling the debugging and guarding operations from the ordinary program execution. Without affecting the original program behaviors, we used the #pragma directives in C language to direct the compiler to handle the NUDA node. The RunAssert compiler will transfer those compounds into binaries, which consist of a configured part and an assertion part. Those binaries are then loaded onto the DCP and NUDA node before target application execution by the RunAssert loader. Once the target program is loaded and is executing, NUDA can monitor the whole system on demand during run-time or can support the generic debugger.

There are two categories of RunAssert directives. The first type is used for parallel program debugging and run-time assertion/guarding. The second type is used for profiling parallel programs. These types of directives can be combined.

Moreover, the directives can be separated into in-place and casual types. In-place directives will be bound with target source code by source symbols and program counters. The RunAssert compiler will link position information onto a demand table that records related information, such as program counters and thread symbols.

#### B. Overview of RunAssert Compiler

The RunAssert work flow is illustrated in Figure 1. According to this flow, users can use the directives supported by RunAssert in the debugging target source code. This work developed a serial toolset to separate the debug directives into code blocks and ordinary source code, and those debugging demand instructions were then translated into NUDA node’s reconfigurable logics. First, the RunAssert code splitter will mark the source code location for those directive code blocks for later compilation and map assignment. The output of the splitter is demand sessions. Second, those sessions contain the users’ debugging issue demands to the target program. The splitter categorizes those sessions and simply reorganizes them at the beginning. The rest of the ordinary code after splitting will be compiled by a generic compiler.

The first step of the RunAssert compiler is to check the syntax of the directives and related assertion expressions in order to avoid the use of incorrect directives or conflicting conditions in directives. The RunAssert compiler must check the semantic logic in debugging directives before progressing further. This work used TCC (Tiny C Compiler) as a basic compiler for compilation of assertion expressions. Essentially, each assertion expression is isolated in the directive blocks. In general, the RunAssert #pragma allows hierarchy declaration, and different #pragmas can be defined to indicate different debugging scopes. If two overlapping debugging scopes define contradicting logic in debugging expression, it would obtain a blind zone and make the run-time assertion functionless.

We notice that, as every symbol’s address is related to a base address, it is necessary to update this base address at the beginning of program execution. We modified the C run-time zero code (CRT0) and updated the base pointer information to NUDA by the CPU’s co-processor instructions. This is indeed an instrument activity, but CRT0 is the absolutely single sequential flow of the whole program. We believe this instrument will not influence SC in parallel programs.

The RunAssert compiler eliminates or merges assertions within the debugging scope. Usually, the debugging expression is simple and without ambiguity. The supporting of directives properly in the coding language is important. This property makes the target source code with debugging/guarding information readable and easy to maintain.

Finally, the RunAssert compiler will generate the target logic for reconfigurable logic in the NUDA node. This work relies on existing synthesizing technology. In other words, RunAssert compiler needs to emit register transistor level (RTL) or higher level languages for synthesis tool to generate netlist file for reconfigurable logic on NUDA node.
C. Run-Time Debugging by RunAssert

The run-time debugging step of RunAssert is as follows. In the view of threading, depending on thread schedules, dispatch strategies and current situations, every thread is unexpectedly executed in each core. For this reason, this work virtualized the shared tables in the NUDA nodes into a large, flat run-time checking table. In addition, for general threading program debugging, the configuration logic in each NUDA node contains the same checking logic by default. In this manner, every thread executing in every core will be monitored without exception.

When the target program starts to execute debugging, the RunAssert loader will load the configuration logic and related information into the NUDA system by ICE or DCP (debug co-processor). With a proper address mapping, we can configure each NUDA node’s reconfigurable logic separately and initialize their internal tables with constraints that are defined by RunAssert. Once NUDA is ready, the RunAssert debug loader will initiate the ordinary generic loader and let the target program be executed. The above steps maintain the integrity of the target program without disturbing debugging demands.

IV. CASES ANALYSIS

Figure 2: Examples of data-race and detected by RunAssert (a) source code with RunAssert (b) RunAssert rules declaration (c) RunAssert compiler generates logics form rules in (b). (①) and (②) are specified during thread creation or context switching, then (③) represents every time access of the shared address, the FSM would check whether race occurs.

This section describes how RunAssert to help to parallel program debug in many ways does.

Acceptable sequence In developing parallel programs, a programmer’s first concern is if the execution sequence is acceptable. Figure 3 illustrates the concept of runtime acceptable sequence checking. The #pragma mdb_configuration declare the customized execution sequence, and the #pragma mdb_anchor is an anchor inserted into the target program. As a matter of convenience, RunAssert directives can group those anchors into a subgroup, Si. Users can specify the execution sequence by anchors or subgroups. In RunAssert, every function name represents an anchor by default. A useful case is grouping by functions initially and then focusing the target in particular subgroups. Finally, users use the anchor directive to identify the demand sequence in the suspect region which was investigated previously.

Race detection Another case of RunAssert is race detection. Figure 2 illustrates the race detection flow using RunAssert from source code to reconfigurable logic on NUDA node. Race condition is one of the most significant and the hardest problem in parallel programs. Because there are huge number of debugging events will spill from each core into the debugging channel. The key issue in detecting race non-intrusively is how to filter the massive suspected memory accesses. With RunAssert directives, users can easily define race conditions to precisely locate the critical section that requires monitoring as well. At beginning, programmer uses the RunAssert directives to identify the target. In this case, we use directive mdb_lock to indicate the locks and use directive mdb_shared to indicate the shared objects. Second, programmer needs to identify the checking rules to objects within the RunAssert global configuration directive. The macro ESR(void (*fp)(void *)) assists users to indicate the exception service routine which is executed as the rules is established. Moreover, the macro LOCK assists users to identify a guarding lock and the following macro MONITOR assists users to list the monitoring shared objects. In order to express the locks hierarchy in the source code, we use the
horizontal expression to identify nested locks. In Figure 2(b), the shared variable X is under nested locks S1 and S2, so the RunAssert represent is LOCK(ThreadA.S1 && ThreadA.S2). The benefit to use the horizontal expression is that users can narrow the monitor scope in the inner locks. For instance, we just need to use LOCK(ThreadA.S2) to describe the process of monitoring the S2. Finally, the RunAssert compiler will translate those directive components into RunAssert logics for RTL translation. The FPGA synthesizer will synthesize the RTL into reconfigurable logic on NUDA node. In this example, the target NUDA FSM shows a simple case, the single lock protection, to capture race.

V. EXPERIMENTAL RESULTS

We developed a target manycore simulator by a parallel simulator mcore [8] in the context of the SPLASH2 benchmarks [9], the related parameters are shown in Table 1. We remain most of the parameters of NUDA, but the target multicore platform is scale down to 16 cores. Because we want to demonstrate a simplest case to show even in such uncomplicated circumstance still have so much performance issues. In the meantime, according to NUDA’s report [2] the area cost of NUDA is about 2.08 mm². And the total area is 564 mm² to the target architecture which is estimated by CACTI 5.3 [10] for the 64 Intel Atom processors, with 16KB I/D cache for each core, and 16MB L2 NUCA for sharing.

Table 2 shows the performance relationship between the RunAssert and pure software debuger GDB. The field memory access in Table 2 shows the memory access count per program, and it is the same the shared memory access field. We assume every shared memory access as a debug event. Consider those debug events are assertions during runtime, without loss of generality, we measure the software assertion in C language and compile with the highest optimization degree. We observe that even the simplest assertion for a range check and output message still need 3–20 instructions. It depends on the assertion expression. If the assertion expression can be decided during compile time, the optimizer will compute the result directly. In contrast, if the assertion expression needs dynamic parameter form memory, the execution time depends on the size of assertion expression.

About the association with memory accesses and debugging events (shared memory accesses and lock events), and the ratio of debugging events is low (2–5%) even in MPEG-4 and H.264. In the performance evaluation, we use the workstation with Ubuntu 8.10 Linux operation system, Xeon EM64T Quatr Core (2.0GHz 12MB L2), 1333MHz FSB, 4GB Memory. We execute that benchmark with a watchpoint to monitor the shared addresses in GDB. The result shows even if watching only one shared memory address, the GDB still needs to compare every memory access event. In contract, RunAssert accomplish all of the assertion checking and verifying in the separate NUDA system. The influence of the additional monitor tasks is limited.

VI. CONCLUSION

In this paper, we have presented a novel, non-intrusive and efficient self-checking programming model for parallel program debugging. We also developed several debugging concept for parallel program debugging. Such as acceptable execution sequential checking and different type, scope, and target for assertion. The result shows this work can achieve non-intrusive and do not affect the original performance.

REFERENCES