On-line Profiling Techniques

Lecture #11: Tuesday, 6 May 2003
Lecturer: Shivnath Babu, David Bloom, Rohit Gupta
Scribe: John Whaley, Jayanth Gummaraju

1 Introduction

On-line profiling refers to a technique for collecting run-time information of a program on the fly in order to decrease the execution time of the program. Information such as basic block frequencies, branch behavior, memory access patterns, and so on, is collected during run time. This information is used by a virtual machine to optimize the program on the fly.

The amount of hardware and software effort in using profile information can vary substantially depending on the implementation. On one end, the profile information can be exclusively used by a dynamic compiler to perform all the optimizations. On the other end, a dedicated coprocessor can be used to use the profile information to reduce the execution time of the program. In this report, we discuss two papers that use profile information extensively. Firstly, we discuss TEST[1], a Tracer for Extracting Speculative Threads in Hydra. TEST uses abundant hardware support to exploit the profile information. Secondly, we discuss Relational Profiling[2] which uses queries (assembly like instructions) and largely software support to optimize programs.

The rest of the report is organized as follows. Section 2 gives a brief summary of TEST and Section 3 presents a brief summary of Relational Profiling. Finally, Section 4 discusses several issues about online profiling that were discussed during the class.

2 TEST: A Tracer for Extracting Speculative Threads

TEST (Tracer for Extracting Speculative Threads) provides a hardware mechanism for analyzing sequential programs with the goal of locating regions with potential thread-level speculation (TLS). This paper presents TEST and shows how it can be used with Hydra, a CMP with built-in TLS support, in the Jrpm (Java Runtime Parallelizing Machine) to provide on-line profile data to mark candidate regions of code for dynamic recompilation into speculative threads.

The current Jrpm system uses TEST to identify loop level parallelism. The two main analyses it performs are load dependency analysis and speculative state overflow anlysis. The load dependency analysis determines dependency arcs between loop iterations by

comparing timestamps on stores and loads to determine if a given STL (speculative thread loop) has dependencies to earlier threads. The speculative state overflow analysis is used to determine if a given STL would be able to fit in the speculation hardware elements. Using the results of these two analyses, speculative threads are chosen based on greatest expected speedup and least likelihood to overflow the speculation hardware.

The hardware implementation of TEST consists of three main components. First, the dynamic compiler must insert annotation instructions into the code, which allow important events to be communicated with the hardware banks. The second component is the hardware comparator banks, which contains the hardware to perform the timestamp comparisons to calculate the critical arcs and the state overflow analyses and store the results into counters. One comparator bank is used to trace on STL, and an array of comparator banks allows for multiple STLs to be traced concurrently. Finally, the store buffers that are used to hold writes during the speculative execution are used during the profiling to hold the timestamp values needed for analysis.

This paper found that the actual speedup achieved with the STLs chosen by TEST closely matched the predicted speedup. The relative speedup is most important (rather than the absolute speedup) when choosing threads to execute speculatively, and TEST did a good job of this in the benchmarks that were run on it. The accuracy and predictability of TEST show promising results for the use of on-line profiling for extracting TLS.

3 Relational Profiling: Enabling Thread Level Parallelism in Virtual Machines

This paper discusses hardware techniques based on a co-designed virtual machine for profiling. They propose a relational profiling architecture (RPA) (assembly language and required hardware) and corresponding relational profiling model (RPM).

The RPM consists of two basic queries: Instruction-based queries, where all events related to a certain instruction are recorded and Event based queries, where all instructions related to a certain event are recorded. In addition, they propose to support hybrid queries. Each query, defined in the RPA assembly language, contains 4 pieces of information: records of information to be collected, the rate of collection, selection criteria applied to records and action to be taken. The type of information collected can be either architectural (PC, Thread ID, operand values) or implementation (fetch/dispatch/issue rates, latency, branch outcome). Actions communicate the information to the VM (e.g. through messages).

The hardware implementation includes a Profile Control Table that stores the PC of the query instruction and the information that is to be collected and is set by the underlying VM. The information collected from the processor pipeline is passed on to the Query Engine, which itself is a 4-stage pipeline that performs the comparisons and actions specified by the query. The limit on the number of instructions that can be

profiled simultaneously is set by the two factors: number of interconnection networks (between the profiling pipeline and the 'main' superscalar pipeline) and the size of the profile buffers that store collected data.

The paper goes on to evaluate the ideal number of each of these elements; It concludes that 4 interconnect networks and 8 buffers meet the demand of most applications — having these many resources causes ¡2% stalls due to profiling.

The strength of this paper is that it proposes an architecture that is easily extensible to more functions - by simply adding more functionality at the VM level. The drawbacks of this paper are that the query sets allowed limit the types of information that can be collected. Further, there is no dynamic mechanism to increase or decrease frequency of profiles during the execution of a program (the authors suggest this as a future extension). Another possibility would be to consider using generic MIPS-style cores in place of the profiling pipeline and service threads. This would allow the extension of this idea to more generic CMPs.

4 Discussion

In this section, we present the various issues that came up during the class discussion session. For clarity and ease of understanding, we present this section in a question-answer format.

- *What is the tradeoff between software and hardware effort in profiling?*
  Profiling can be implemented in several ways - with a simple query engine and lots of software support, or with abundant hardware support (having a full coprocessor, for example). In particular, relational profiling uses a simple query engine with software feeding in explicit instructions, and TEST uses abundant hardware support (comparator banks for dependence analysis between threads, for example).

- *What types of data sets should be used for profile-based optimizations?*
  Many people try to do profile based optimizations and use exact same data set for both the training and the evaluation of their technique. You should have at least two different data sets, and preferably more.

- *If you are a compiler writer or an architect, what profile information would you like to know?*
  Basic block frequencies, length of dependencies, branch direction, data access patterns (sequential/strided; local/remote) how long between I/O, access to remote versus local memory to save communication, how often we access specific memory, detect memory conflicts, and synchronization issues, such as whether to use a spin lock and the number of iterations a spin lock typically executes. Also the instruction mix, number and ratio of hits to misses in the cache, different types of misses in the cache, memory disambiguation (similar to TEST) value profiling.

- *How much hardware support is needed to support profiling?*

  One problem is that the more hardware you have, the more profile information you will have, and then managing the profile information can become a problem. There is a cost associated with getting better profile data. We need to include a mechanism to say when we want to trigger the software based on the profile information. The software might become a bottleneck or drop information. One solution is to put the software task completely into hardware. Another solution is to still have software, but try to summarize the information in hardware before passing it to the software.

- *How can static compiler help in online profiling?*

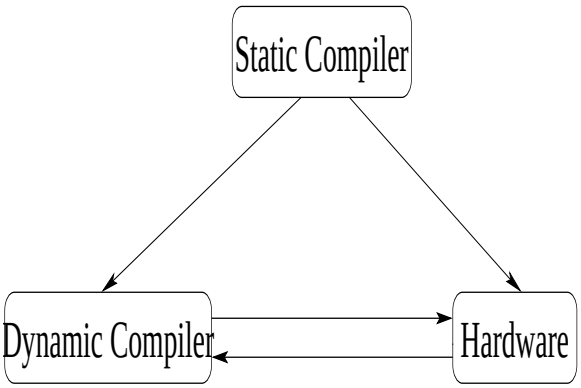

Figure 1: Hardware software interaction for profiling.

  A static compiler can communicate to the dynamic compiler the optimizations that it either failed to perform or performed conservatively. The dynamic compiler, with the run-time profile information, can use the information provided by the static compiler to perform more aggressive optimizations. This especially works out well because not only is the time for performing dynamic compilation reduced, the static compiler can give information corresponding to the source code. Figure 1 illustrates this interaction.

- *How can the association of events with locations in the program be made? For example, for which region of memory was there a cache miss? At which program counter was there a cache miss?*
  An inexpensive way of getting and summarizing information about memory and branch behaviors is to periodically dump the cache tables and branch prediction tables. Hardware probably also already includes a good summary of memory access patterns (stride, etc.) through prefetch hardware, etc. We could utilize this hardware for gathering and summarizing profile statistics.

- *What are the issues in collecting profile information with respect to the changing behavior of the program?*
  When collecting profile data, it is important to be aware of different phases, especially the initialization phase. It is probably a good idea to turn off profiling at the beginning of the program because there is a lot of noise during initialization. We can utilize the hardware to help detect phase changes. The hardware can identify dramatic changes in branch predictor miss rate, cache miss rate, etc. and flag these as phase changes.

- *Can the virtual machine automatically find the bottleneck in the program? Also, how can you associate the code with the profile data?*
  TEST is clever about this. Not only is there a problem, TEST tests whether or not it is worth solving. If there is a problem with associativity, we can switch to another associativity. In general, a "What-if" analysis is very hard to predict. It is relatively easy for systems like databases, but machines are complicated. It is a general optimization problem, and the model is extremely hard with many variables. The hardware configuration space is small, so the problem may be more manageable. Doing this in general for compilers is much harder.

- *What is the role of VM during profiling?*

  The hardware can determine what is important. One technique is to start the VM by saying "tell me events that are frequent", then ask "is my optimization useful or not?" Another technique is to use information from the static compiler. The static compiler can direct the hardware to validate its decisions. The hardware communicates information to the dynamic compiler, and the dynamic compiler can make changes to the code. Dynamic compilers typically insert run-time overhead, and using the information from the static compiler is one way of reducing this overhead.

- *Is it possible to obtain more information than what can be obtained via cycle-accurate simulation?*
  One problem is that I/O, OS, etc. are hard to get accurate. We can also do continuous profiling on actual applications, and the user can use the profiler easily. Another approach is to use a VMware-like solution to profile more easily. (Relational profiling uses this approach.) We need a virtual machine for dynamic compilation, anyway.

- *What are some of the issues for developing a profiling infrastructure for CMPs?*
  The first question is what we want to get out of profiling. We can use it to identify parallelism or effective (or ineffective) speculations. We can also use it to dynamically reconfigure the CMP to match the characteristics of the application. What can we profile? A couple possibilities are inter-thread dependencies, memory access patterns.

On a CMP, we could offload profiling work onto a service processor. This is true in general: even OS stuff like network stack, etc. could benefit from this. Some processes don't need the full resources.

If we want to use some processor as a profiling processor, we would need some connections between the processors. One technique is for a processor to bundle up information and write to memory, and then signals another processor. It could use local memory for quick communication.

This brings us to the idea of heterogeneous CMP architectures. One example would be having many types of Alpha processors on a single chip: one chip 8-way, 4-way, 2-way, pipelined 1-way, simple 1-way. The scheduler could then schedule each application on an appropriate processor. This may lead to power efficiency. Playstation 2 used this idea; the I/O controller of the Playstation 2 was the same as the main CPU of the PS1.

- *When can we consider a profile to be "accurate"? In other words, what are the convergence metrics for profiling?*
  The online profile information has to be collected repeatedly as the behavior of the program might change dramatically over the course of program execution. The issue then is to tune the sampling time and sampling frequency such that adequate profile information is obtained. Collecting more than necessary information can induce huge overheads potentially affecting performance. However, there is no clear answer to this question. One obvious metric is: When using the profileinformation, how much optimization do you get?

- *How often should the profile information be collected?*
  If you do periodic sampling like the relational profiler (every 256 cycles), you can miss periodic events, or not have enough granularity. One solution to this is to do statistical variation on the sample period.

References

[1] Chen, M., and Olukotun, K. Test: A tracer for extracting speculative threads. In *International Symposium on Code Generation and Optimization* (March 2003).

[2] Heil, T., and Smith, J. Relational profiling: Enabling thread level parallelism in virtual machines. In *International Symposium on Microarchitecture* (December 2000).