Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.

Ideas on improving Linux infrastructure for performance on multi-core platforms

Maxim Alt
Intel Corporation
maxim.alt@intel.com

Abstract

With maturing compiler technologies, compile-time analysis can be a very powerful tool for optimizing and monitoring code on any architecture. In combination with modern run-time analysis tools and existing program interfaces to monitor hardware counters, we will survey modern techniques for analyzing performance issues. We propose using performance counter data and sequences of performance events to trigger event handlers in either the application or the operating system. In this way, sequence of performance events can be your debugging breakpoint or a callback. This paper will try to bridge the capabilities of advanced performance monitoring with common software development infrastructure (debuggers, gcc, loader, process scheduler). One proposed approach is to extend the run-time environment with an interface layer that will filter performance profiles, capture sequences of performance hazards, and provide summary data to the OS, debuggers, or application.

With the introduction of hyper-threading technology several years ago, there were obvious challenges to look beyond a single running process to monitor and schedule compute intensive processes on multi-threaded cores. Multi-level memory hierarchy and scaling on SMP systems complicated the situation even further, causing essential changes in kernel scheduler and performance tools. In the era of parallel and platform computing, we rely less on single execution process performance—with each component optimized by the compiler—and it becomes important to evaluate performance of the platform as a whole. The new concept of performance adaptive schedulers is one example of intelligently maximizing the performance on platform level of CMP systems. Performance data at higher granularity and a concept of processor efficiency per functionality can be applied to making intelligent decisions on process scheduling in the operating system.

Towards the end, we will suggest particular improvements in performance and run-time tools as a reflection of proposed approaches and transition to platform-level optimization goals.

1 Introduction

This paper introduces a series of ideas for bringing together existing disparate technologies to improve tools for the detection and amelioration of performance hazards.

About 20 years ago, an exceptionally thin 16-bit real-time iRMX operating system had an extremely simple but important built-in feature: when debugging race conditions in shared memory at any point of run-time execution, a developer could bring the system to a halt, set an internal OS breakpoint at an address and the operating system would halt whenever a value was being written at the specified address. No high level debugger was needed to debug race conditions, nor were they capable.

With the introduction of hyper-threading technology, many software vendors started to experiment with running their multi-threaded software on hyper-threaded processors. With the increase the number of processors, those vendors expected to get good scaling after elimination of synchronization and scheduling issues. These issues are quite difficult to track, debug, or find with Gdb or VTune analyzer.

The problem of debugging synchronization issues in multi-threaded applications is growing more important and more complex with the advent of parallelizing compilers and language support for multithreading, such as OpenMP. The compiler has knowledge of program semantics, but does not generally have run-time information. The OS is in a position to make decisions based on run-time information, but it doesn't have the semantic information that was available to the compiler, since it sees only the binary. Further, any run-time analysis and decision-making affects application performance, either directly by using CPU time, or indirectly by effects such as cache pollution.

Spin locks are an example of the kind of construct for which higher-level information would be helpful. The compiler can easily detect spin lock constructs using simple pattern-recognition. From the run-time perspective a spin lock is a loop that repeatedly reads a shared memory address and compares it to a loop-invariant a value. A spin-lock is a useful synchronization mechanism if the stall is not long. For long stalls, it wastes a lot of processor time. Typically, after spinning for some number of cycles, the thread will yield to let other threads make progress. If the OS could identify which threads were waiting for a lock and which threads held the lock, it could adjust priorities to maximize throughput automatically.

Another relevant example of how a scheduler might use performance data is not based on debugging. Consider two processes running concurrently: two floating point intensive loops, where only one of which has long memory stalls. Should the scheduling of the processes alter? For example, two floating point intensive loops with unknown memory latency or two loops of unknown execution property or two blocks of code of unknown programming construct?

This question was raised by my colleagues in [1] about 3 years ago, where it was discussed how beneficial it would be if the OS scheduler had built-in micro-architectural intelligence.

Another issue with debugging the performance of an application using spin locks is that it typically doesn't provide much insight to know that the spin-lock library code is hot. The application programmer needs to know which lock is being held. That information can be gathered from the address of the lock, but often it is more useful to have a stack trace gathered at the time the lock is identified as hot. This requires sampling the return stack in the performance monitor handler, not just the current instruction pointer.

A process or a code block (hot block or block of interest) can be characterized by performance and code profiles, where the code profile is represented by a hierarchy of basic programming constructs, and the performance profile is represented by execution path along with registers image captured at any given point in time. In this paper we will describe a set of new profile-guided static and dynamic approaches for efficient run-time decisions: debugging, analyzing and scheduling.

Please refer to Appendix A for an overview of existing technologies, tools and utilities.

2 Bridging the Technologies

I would like to explore extending the scope of sampling-analysis hybrid tools for example by profiling with helper threads[10.5].

This section will provide a series of examples on how to combine building block components to get useful sample-analyze schemas, which could potentially turn into standalone tools:

- *A Pintool doing Pronto.* Given the non-existent ability of Pin to sample performance counters in optimize/analyze mode, a hybrid tool when Pronto is based on a Pintool would allow dynamic implementation of collecting performance data via pintool instrumentation. If sampling is needed then a sampling driver can be initiated and called from a pintool using PAPI. The PAPI interface allows you to start, stop, and read the performance counters (e.g. calls to PAPI_start and PAPI_stop using Pin's instrumentation interface). PAPI does require a kernel extension. This idea may also be implemented through a static HP's Caliper tool[1]

- *A Pintool to seek for hotspots.* Hotspot analysis can be done by defining "what it means to be a hotspot" by a pintool, or statically by parsing sampling data with scripts.

- *A Pintool which uses performance feedback data.* Theoretically, a Pintool could be built which uses performance data which has been collected (perhaps by some other tool) on previous runs. Intel has a file format for storing performance monitor unit data (".hpi" files) which are used by the compiler. Pintool reads events and performance counters dynamically as it executes a binary.

- *A Pintool to recognize event patterns.* Sequitur can be used for static or dynamic analysis (with a certain performance overhead) for complex grammars or in the context of this paper, event sequences.

2.1 Performance Overhead of *Sample-analyze* Workflow

Pintools instrumentation can be intrusive and the overhead is dependent upon the particular tool used. The generated code under pintool is the only code that executes, and the original code is kept for reference. Because instrumentation can perturb the performance behavior of the application, it is desirable to be able to collect performance data during an uninstrumented run, and use that data during a later instrumentation or execution run.

In addition, Sequitur performance for generic grammars containing many symbols may be extremely heavy[2].

2.2 Developers' Pain

As one could imagine, the multi-threaded application developers who are debugging and running on multi-core architectures need profiling tools (such as VTune analyzer or EMON) to be aware of the stack frame and the execution

[1]`http://h21007.www2.hp.com/dspp/tech/tech_TechSoftwareDetailPage_IDX/1,1703,1174,00.html` (currently available for Itanium microarchitecture only)

[2]Incorporation of the Sequitur algorithm into your instrumentation is an essential part of the techniques described in this paper. Due to the significant performance overhead, Sequitur is not used in generic form, and requires tailoring for particular usage cases with simplified grammar. It helps to find performance hazards and sequences of events of interest, or these, which characterizes the application process.

context. The tools also need to take into account procedure inlining. It would also be useful if the simple data derived out of these tools could be used by the run-time environment to adapt the environment for better throughput.

In this section we would consider code examples known to cause much pain when debugging or scheduling. Then, we will suggest ways to adjust the Linux infrastructure to leverage and integrate existing tools mentioned above to address these painful situations.

2.3 Profile-guided Debugging

A very common example is when you profile a multi-threaded application with frequent inter-process communications and interlocking. Many enterprise applications (web servers, data base servers, application servers, telecommunication applications, distributed content providers, etc.) suffer from complex synchronization issues when scaled. While optimizing such application with standard profiling tools, it is common to observe most of the cycles being spent in synchronization objects themselves. For example, wait for an object, idle loops, spin loops on shared memory.

Whether the implementation of synchronization objects is proprietary or via POSIX threads [15], a hot spot is noted as entering/leaving the critical section or locking/unlocking the shared object. Deeper analysis of such hotspots usually shows there is not much to optimize further on a micro-architectural level unless one is trying to optimize the performance of glibc. The real question is how to find out what objects actually originated a problematic idle or spin time in a millions-of-code-lines application with hundreds of locks? To track and instrument locks is not an easy task; it is similar to tracking memory allocations. Standard debugger techniques are not effective in identifying the underlying application issue.

```
Spinlock (mutex_t *m) {
Int I;
For (i=0; I < spin_count; i++)
  if (pthread_mutex_trylock(m) != EBUSY) return;
  pthread_mutex_lock(m); //or sometimes Sleep(M)
}
```

Figure 1: Spin lock

```
spin_start:
  pause
  Test  [mem], val  ; pre-read to save the
                    ; atomic op penalty
  J     Skip_xchg
  Lock  cmpxchg [mem], val ; shows bus serial-
                           ; ization stall

Skip_xchg:
  jnz   spin_start
```

Figure 2: Spin Lock Loop

A standard adaptive spin lock implementation looks similar to Figure 1 where inner spin lock loop translates to instructions shown in Figure 2.

Let's analyze what characterizes this code. One obvious implication of using atomic operations (for entering/leaving critical section it is atomic `add/dec`) is that such operations serialize the memory bus, which yields significant stalls due to pipeline flush and cache line invalidation for `[mem]`.

The code in Figure 2 would generate similar performance event patterns on most architectures. Following are the properties which characterize the code block profile:

- Very short loop (2–5 instructions)
- Very short loop containing `nop` or `rep nop` (`pause`)
- Contains instruction with Lock prefix yielding bus serialization
- Contains either `xchg` or `dec` or `add` instruction

The performance event profile for this block has the following properties:

- Likely branch misprediction at the 'loop' statement
- Very high CPI (cycles per instruction), as there is no parallelism possible
- Data bus utilization ratio (> 50%)
- Bus Not Ready ratio (> 0.01)
- Burst read contribution to data bus utilization (> 50%)
- Processor/Bus Writeback contribution to data bus utilization (> 50%)
- Parallelization ratio (< 1)
- Microops per instruction retired (> 2)
- Memory requests per instruction (> 0.1)
- Modified data sharing ratio (> 0.1)
- Context switches is high

We can define the grammar which consists of: loop length, loops with nops, locks, adds, dec, xchg; misprediction branches, high CPI, context switches, high data bus utilization.

It is necessary to quantify each of the performance counters' values (also called knobs) so we could establish a trigger for potential performance hazard. From the properties of spin lock block we can define a rule for a block to become the hot block or the block of interest. Then, similar to hot stream data prefetch example in Appendix A, we will use Sequitur to detect hot blocks containing event sequences within the defined grammar as you can see below in Figure 5

In order to simplify the problem for the spin lock detection, we may limit ourselves to the analysis of only code profile, as the entire performance profile is a direct result of having an instruction with a 'lock' prefix (e.g. a 'lock' prefix on any instruction results in data bus serialization, yielding known set of performance stalls). The 'spin lock' code properties can be dynamically obtained at run-time by instrumentation. We can use the Pin command line knob facility to define a block's heat. For example these knobs may be:

- Matched number of samples to trigger the hazard
- Number of consecutive samples
- Minimum and maximum length for hot block
- Minimum spins to consider it hot
- Maximum number of instructions to consider loop short

This technique[3] would allow us to insert a breakpoint on an event of performance hazard - the hot spin lock according to user's definition of a performance concern. The debugger would be able to stop and display stack frame of the context when the given sequence of events had occurred.[4]

The described dynamic mechanism (one of the suggested "sample-analyze" workflows) detects a block of interest and breaks the execution on a performance issue.

[3]For the spin lock profile, running the sampling along pintool instrumented executable is safe and correct, since main characteristics of spin lock could not be disrupted by instrumented code shown above

[4]Ideas for a standalone profiling tool - Assume we have an ability to improve an open source (PAPI-based) or proprietary (VTune analyzer) profiling tool. Instead of Int 3 insertion we could insert a macro operation to dump a stack frame and register contents by the sampling driver. The existing symbol table would allow tracking source-level performance hazards defined by the pintool's knobs.

```
// Run in optimize mode only - no need for sampling mode run

for (each basic block)
  for (each instruction in block) {
    if (Instruction is branch) {
      target = TargetAddress(ins);
      if (trace_start < target && target < address(ins) &&
         (target - address < short loop knob)) {
        Insert IfCall (trace_count--);
        Insert ThenCall (spin_count++);
        New grammar (knobs);
      }
    }

    if ((instruction in block has lock prefix) &&
       (instruction is either xchg, cmpxhg, add, dec)) {
      if (grammar->AddEvent(address(ins)) &&
         (block_heat++ > hot block knob)) {
        Insert Interrupt 3;        // for debugging the application
      }
    }
  }
```

Figure 3: Pintool's trace instrumentation pseudocode

For the static profiling schema we will modify this workflow as follows:

On run-time side, as follows:

- Pintool instrumentation inserting software-generated interrupts would stay the same as in dynamic case. Pintool would read the information about hot spin locks from static profiling results (PGO) or pronto repository

-Allow the debugger to read pronto repository directly. This data would contain pairs, such as (ip address, number of times the IP is reached - signifying the heat of the block)

On compile-time side:

- A newly developed pintool that would be similar in functionality to PGO and profrun utility without sampling. However this pintool would contain same detection algorithm by the Sequitur as described in Figure 3, which detects hot spin locks according to user defined knob values marking the heat. In this manner, pronto_tool is virtually replaced with the Sequitur. Upon detection of a hot block, the pintool spills the pair (ip, frequency) into the pronto repository.

- Due to unique code properties, the current implementation of PGO and profrun utility already contain the needed information about spin lock block's code profile. We still need to build a script which would replace analysis tool pronto_tool and is based on parsing profile data with the Sequitur. This mechanism would extract event patterns matching our definition of the spin lock block's heat. The detected pair (ip address, number of times this IP has been reached until block became hot) is inserted into profiling info, and subsequently passed to the debugger

This would summarize another suggestion on a new standalone run-time tool that reads in the profile data and use it to find hot locks, and then tells the debugger the IP of those blocks so it can stop there.

With our attempt to characterize code by its performance profile, one may ask how adequate the mapping between an actual code block and its performance profile is. Would a sequence of events spanned by performance properties symbolize a spin lock code, or in other words, how uniquely do code block properties define the code itself? For performance debugging or adaptation the functionality of a hot block itself is not important. Rather, what important is it's algorithm mapping on the micro-architecture and the stalls caused by this mapping. Therefore, it is sufficient to accurately describe a performance hazard and signal when its properties have occurred.

2.4 Performance Adaptive Run-time Scheduler

Consider running a high performance multi-threaded application. Many computation and memory intensive applications (rendering, encoding, signal processing, etc.) suffer from complex scheduling issues when scaling on modern multi-threaded and multi-processor architectures. Often, the developers optimize these applications by parallelizing single threaded computation to run multiple threads. In order to make the application run well in parallel, the developers perform functional decomposition.[5]

Then, the OS scheduler takes over the decision on how to schedule these functionally decomposed threads onto the available hardware. Since the OS scheduler is not aware of micro-architecture, functional decomposition, or OpenMP, parallelization often leads to performance degradation. In order to analyze this phenomenon there were many researches on informed multi-threaded schedulers [12], symbiotic job scheduling for SMP [13], [14], and MASA [1]. In this paper we will take an approach of bridging existing profiling tools and advanced compiler technologies to take a step further in solving this problem.

As an example, consider open source *LAME mp3 encoder*[6]. It is clear that the application is both computation and memory intensive, where computation is mostly floating point. Functional decomposition of hotspot function *lame_ecnode_mp3_frame()* is equivalent to a functional decomposition of *L3psycho_anal()* function. All decomposed threads at any point in time could unfold into a situation when running processes utilize similar resources on the same physical core (e.g. threads are: floating point intensive, floating point intensive, heavy integer computations, heavy integer computations, long memory latency operations, long memory latency operations).

As in the previous section, running a thread's profile consists of performance and code properties. Below, we will analyze such properties and the knobs defining the heat:

Following is the structure of properties for floating point operations intensive code block:

- Estimated functional imbalance originated by compiler's scheduler

- Estimated CPI by the compiler's scheduler

- Outer loop iteration count

[5]Functional decomposition is the analysis of the activity of a system as the product of a set of subordinate functions performed by independent subsystems, each with its own characteristic domain of application.

[6]`http://lame.sourceforge.net/download/download.html`

You can see similar characteristics in integer intensive and memory intensive code blocks. These code block properties ignore possible coding style inhibitors and are agnostic to some optimization techniques (such as code motion). Nested loops and non-inlined calls within a loop are being merged into single region at the run-time, since the block of interest in this case would be an outer block encapsulating multiple iterations to the same instruction pointer.

Event profile to determine performance properties for floating point intensive block:

- Balanced execution and parallelism – actual cycles per instruction ratio

- Microops per instruction retired for very long latency instructions (FP)

- FP assist and saturation event per retired FLOPs

- Retired FLOPs per relative number to instruction retired

- Conversion operations RTDC per relative number to instruction retired

- SSE instruction retired per instruction retired

Event profile for memory intensive block:

- Data bus utilization ratio (> 30%)

- Bus Not Ready ratio (> 0.001)

- Burst read contribution to data bus utilization (> 30%)

- Processor/Bus Writeback contribution to data bus utilization (> 30%)

- Microops per instruction retired (> 2) for repeat instructions

- Memory requests per instruction (> 0.1)

- Modified data sharing ratio (> 0.1)

In order to write a pintool for this topic, it is necessary to be able to deliver some compile-time derived data to the run-time. In particular, some code block characteristics can be easily determined by the compiler's scheduler: For example, CPI and other parallelism metrics, scheduled memory operations, scheduled floating point operations, etc. Compilers can output such information via object code annotations, optimizer reports, or post-compilation scripts that can strip required statistic on the generated assembly code. The recent changes in GCC's vectorizer and optimizer include Tree SSA[7]. It is possible to get the compiler scheduler's reports using `--ftree-vectorizer-verbose` compiler option. The code block properties derived from the compiler scheduler's data only needed on hot blocks. However, the compiler does not know which block is hot unless PGO or Pronto was used.

On the run-time side, Pin has a disassembly engine built-in. A pintool would be able to easily determine functional unit imbalance in a hot block if it isn't available from the compiler. Assuming that Pin has the ability to retrieve some compiler scheduler data, there are a few ways to create a pintool to determine whether running code has properties of floating point or memory intensive hot blocks:

For the compile time:

- A "2-model" compilation can usually do both: determine and process hot block properties. Generated assembly, PGO and Pronto repository can be concurrently processed with a script to extract the instruction level parallelism (ILP) information per hot block

[7]Static Single Assignment for Trees [18]: new GCC 4.x optimizer: `http://gcc.gnu.org/projects/tree-ssa/\#intro`

- Extend the code's debug-info into information containing ILP of basic blocks during compilation. It is an estimated value, not based on run-time performance. The scheduler's compile-time data could be passed through an executable itself as a triple (start block address, end block address, parallelism data). Pintool has an extensive set of APIs that accesses debug info.

For the run-time:

- Instrument the binary with a pintool that traces loops with a large number of counts (a potential knob). Then, count the number of floating point, memory and integer operations in a loop.

- PGO and Pronto may also contain ILP related ratios, which are derived from basic sampling during profiling run (with PAPI interface). Extending the Pronto repository to carry parallelism info can improve the ability of pintool's instrumentation analysis.

Additional run-time instrumentation can be based on the performance profile of the hot computational intensive blocks by running sampling along with instrumentation.[8]

This schema shows the feasibility of obtaining a process property. However, possible performance overhead of sampling and processing (even if it is incorporated in one instrumentation-sampling step) may be too heavy to make run-time decisions for the OS scheduler.

Now we will analyze the data collection process for the performance profile-aware OS scheduler. First, let's exclude 2-step models as inappropriate schemas for OS scheduling. Assume the OS cannot contain low overhead continuous sampling, then, a pintool instrumentation embedded into a running process cannot be extensive but can be discrete.

We will also assume that estimated ILP information and compiler scheduler's data can be retrieved via debug information for each basic block[9]. Instrumentation can count frequency and count of each basic block determining the estimated heat of the block.

On the run-time side, we would require implementation of one of the following: limited sampling, processing of Pronto repository, or decomposing compiler scheduler decision for the length of one basic block. We propose that context switch time might be an appropriate place to insert this lightweight process.

Pin instrumentation can be done on a basic block granularity with Pin itself setting up instrumentation calls, which is greatly improves performance.

Thus, we are considering three approaches for dynamic performance adaptive scheduler:

1. Annotation, no instrumentation. A limited lightweight instrumentation is done only on the level of basic blocks. This instrumentation would not be based on performance counters, clock cycles or actual ILP info. This is assuming some basic compiler scheduling data can be incorporated in to an executable using mechanisms similar to debug symbols. This would provide an estimated code block profile. As soon as a code block gets to a specified heat (user pre-defined knob on loop iteration count), the pintool triggers an internal OS scheduling event carrying the code profile signature.

2. Limited sampling. As noted earlier, limited sampling may be possible at the OS scheduler's

[8]The unique performance profile reflecting compute intensive block properties would not be disrupted by instrumentation performance overhead

[9]The Pronto data is mapped using debug info in DWARF2 format. Some compiler-based info such as predicted ratios IPC or FPU/ALU/SIMD utilization could be added to pronto repository data derived from pre-characterized hot blocks

checkpoint, such as context switch. This could refine information obtained from item 1 above and give more accurate data on actual ILP. A single sampling iteration over basic block could detect performance profile hazards based on counters which are specific to compute intensive blocks shown above. A trial sampling run would last only during the length of a single iteration of the loop, assuming a context switch had occurred several times during execution of a large loop count.

3. Instrumentation under "2-compile" model. Assuming ILP information can be incorporated into the binary, we would use PGO or Pronto mechanisms to generate actual sampling ratios within the profile feedback repository. After a training run we collect the profile information which includes each basic block's frequency and count, along with its ILP info. Assuming this information is available in the binary, this would indicate to the OS scheduler the performance properties of the running process. This workflow would be enabled by a simple pintool instrumentation that is analyzing each basic block's information.

From the workflow above, there is an obvious conclusion that the loader/linker has to have certain abilities to map and maintain new information passed within the generated binary. Investigating the glibc code on potential changes for loader/linker in *elf/dl-open.c*, *dl-sym.c* and *dl-load.c,* we noted a possibility of creating a number of loading threads that could load libraries in parallel. With *_dl_map_object_from_fd()* each of the threads would retrieve various information carried in the executable by link-time procedure of locating symbols. In this way hashing mechanism for objects with large amount of symbols can be parallelized in *dl_lookup_symbol_x()*, calling the expensive hashing algorithm *do_lookup_x()*. However, conducting this experiment any further is out of scope for this paper.

It is appropriate to comment on describing possible workflow combinations of "sampling-analysis" of the static algorithm for the OS performance adaptive scheduler. Due to the nature of the usage model, the static algorithm may be suitable for feasibility study or prototyping an approach, but least likely used in real life and therefore is not mentioned in this paper in detail.

Each of the workflows discussed require certain capabilities to be developed:

1. Sampling drivers are closed source but can be distributed. The open source interface for TB5 format analysis should be implemented by PAPI or VTune Analyzer/SEP.

Most of the performance events required for determining the code's performance profile are public.

2. Compiler (GCC). The performance analysis tool with basic profile feedback and vectorizer reports mechanisms already exist. The following enhancements would be needed:

- The vectorizer reports must include compiler scheduling information on parallelism.

- Mechanisms to incorporate compiler reports per basic block in to a binary need to be developed.

- A utility which collects sampling data for profile feedback needs to be developed based on the PAPI interface.

3. Pronto repository and profrun utility. These utilities currently exist as a part of Intel Compiler, but are closed source because they use the VTune TB5 file format. The following enhancements would be needed:

- PAPI interface for profrun utility and Pronto repository

- Pronto using pintool instrumentation

4. Pin. The following enhancements would be needed:

- API extensions to retrieve compiler scheduler info that are embedded into a binary

- Compiler scheduler decomposition API (an APIs that retrieve compiler's scheduler information, especially related to ILP)

- API ability to read Pronto repository from memory or a file

- A 'timer' pintool to help development activities to track performance (via gettime())

- New pintool instrumentation libraries to provide description of hazardous performance event sequences based on common code and performance profiles

- New Pin APIs that can perform independent sampling via PAPI interface to hide architecture dependences, "A Pintool doing Pronto"

- For each pintool instrumentation to specify a set of performance counters that may be affected by instrumentation itself.[10]

5. Loader/linker. The loader can be easily instrumented with the Pin interface relying on IMG_ API set. Following enhancements would be needed:

- Properly dispatch additional compiler's scheduler information embedded into binary, similarly to the debug info

- For faster linking, improve the OS loader's speed by creating loading threads

[10]These performance counters or ratios may not be present on your architecture with specified name but on modern architectures assumed to have similar ones

6. VTune analyzer. Following enhancements would be filed to Intel VTune development team:

- Ability to recognize and sample pintool instrumented code.

- Capability to receive a signal from instrumented code in order to display and translate process context and stack frame.

7. Debugger (GDB):

- Compile-time feedback: Enable reading basic block ILP information along with debug information incorporated by the compiler's scheduler

- Run-time feedback: Enable reading Pronto repository with (frequency, count) information of the BBL. It may eliminate the need for pintool instrumentation for the debugger.

- Consider scripting language to describe event sequence and pintool instrumentation algorithms.

8. OS Scheduler:

- Need to have the ability to retrieve process profile signature which characterizes performance and code constructs derived from running executable.

In order to test the feasibility of the suggested tools without changes in the kernel, we can write emulation application with the user mode simplified scheduler's algorithm by setting affinity with process's profile data.

- Have the option of signaling to the pintool instrumentation process that a context switch is about to occur and start a lightweight instrumentation mechanism with non-intrusive sampling for one iteration of a basic block. Pintool may communicate with the OS scheduler via ioctl, considering the events are coming from a driver.

3 Conclusion

In this paper we surveyed ideas spanning several technologies and tools developed by a vast community during past 10 years. We tried to bridge recent accomplishments in mainstream compiler technology, performance counters, pattern recognition algorithms, advanced binary instrumentation tools, debugging approaches and advanced dynamic optimization techniques. Many of these technologies were also inherited from previous researches on databases optimizations and compression algorithms. Demonstrated complex workflows incorporating "sample-analyze" technologies into enhanced run-time Linux infrastructure make another step towards advanced dynamic optimizations, debugging and process scheduling. Quantifying the significance and usefulness of the proposed approaches is a subject of a separate research and experiments.

Please refer to Appendix B for potential applications of the suggested ideas.

4 Acknowledgements

I would like to thank my colleagues and highlight their involvement in supplying important supporting materials, thoughts and comments:

Siddha Suresh, Venkatesh Pallipadi: Intel, Open Source Technology Center

John Piper, Anton Chernoff, Shalom Goldenberg: Intel, Dynamic Optimization Labs;

Robert Cohn, Harish Patil: Intel

Raymond Paik: Intel, Channel Product Group

5 Appendix A. Technology Background and Current Situation

Most modern micro-processors have a PMU—virtual or physical performance monitoring unit that contains hundreds or even thousands of performance counters and events. Modern micro-architectural profiling technology is divided into two distinct steps: sampling and analysis. The sampling mechanism records instruction pointers (IP) with performance counters as in (IP, frequency, count) or (IP, value). The analysis processes sampling data on the maximum time spent in repeated blocks (hot blocks), possibly including: disassembly, mapping to the source code, affiliating to a function, a process or a thread;

As the building blocks of the proposed workflow, it is important to overview existing tools and technology. Some of these tools are Open Source, some are proprietary or have a closed source engine with a BSD-style license for free distribution.

Sampling tools:

Well-known profiling tools and programming interfaces (such as VTune analyzer, EMON, PAPI, Compiler's profile guided optimization (PGO) with sampling) are usually system-wide and process agnostic. Sampling tools can be attached to any running process, but do not have access to full run-time environment and functionality context: thread storage, register values, loop count, frequency and stack.

Another type of sampling tool does not have the concept of time and is built upon executed instructions along the execution path. Such tools are not aware of stalls and clock cycles, but can sample executed instruction properties such as instruction count, instruction operands, branches, addresses, functions, images, etc.

Pin [8], [17] can be considered as a "JIT" (just-in-time) compiler, with the originating binary execution intercepted at a specified granularity. This execution is almost identical to the original. Pin contains examples of instrumentation tools like basic block profilers, cache simulators, instruction trace generators, etc. It is easy to derive new Pin-based tools using the examples as a template.

Analysis tools:

Well known analysis tools are compilers and debuggers. Beyond actual code generation and instruction scheduling, the compiler has the ability to report on optimizations, scheduler heuristics and decisions, and predicted performance counter ratios (please refer to [5]). The compiler can determine code profile and estimate performance profile of any code block, specifying execution balance across CPU units.

Some sampling tools such as VTune and EMON incorporate data analysis within them.

There are advanced parts of a compiler's optimizer which can be standalone tools that are able to parse sampling files and extract profile data.

Before moving onto another class of tools, there are also more sophisticated data analysis tools which work in formal language environments. As *Sequitur* originated from compression algorithms, it is capable of defining a sequence of events as a grammar and trace event samples on correct expression composition from given sequences of samples. The Sequitur algorithm description can be found in [9]. The implementation of sequitur has public versions.

Hybrid tools:

The series of following tools are a hybrid between sampling and analysis. Most require a complex build model with up to 3-compilation model process, (see Figure 5).

Profile-guided optimization (PGO for Intel Compilers or Profile feedback for GCC) is a part of many modern compilers [3], [4]. Currently, the PGO mostly samples executed branches, calls, frequency and loop counts with following output of the data in an intermediate format to the disk. The PGO analysis is also a part of compiler's optimizer, which is invoked with second compilation. It assists the compiler's scheduler to make better decisions by using actual run-time data instead of heuristics targeting probable application behavior.

As an extension of the PGO mechanism, a tool incorporating a trace of run-time specific events and samples (for instance, actual memory latency, cache misses) has been developed. This mechanism can be considered as a bundled sampling tool of *profrun utility* with analysis tool *pronto_tool* [6], [16], which we will refer to as Pronto. Rather then just being a natural extension of PGO capabilities these tools are standalone and not incorporated into the compiler. Architecture-wise, profrun is built upon proprietary sampling drivers spilling the data on the disk, which is called *Pronto repository*. Profrun is currently incorporated in the Intel Compiler package using Intel sampling drivers, but conceptually can be based on open source *PAPI interface*, see [7] for PAPI documentation. The pronto_tool reads and analyzes the Pronto repository for various data representation. A typical output is shown in Figure 4.

A hybrid tool *Pintools*, based on Pin, is a critical component of this paper's focus. Pintools incorporate into a single executable targeted instrumentation and analysis. This is an implementation powered by Pin API callbacks providing instrumentation for any running image at any granularity. Pintools mechanism can be considered a generic binary instrumentation template to create your own hybrid of sampling

```
$ profrun -dcache mark
$ pronto_tool -d 10 pgopti.hpi

PRONTO: Profiling module "mark":
PRONTO: Reading samples from TB5 file 'pgopti.tb5'
PRONTO: Reading samples for module at path: 'mark'

Dumping PRONTO Repository

Sample source 0: pgopti.tb5 UID: TYPE = TB5SAMP (54423553 414d5000
80ac9d3c 8f39c501 0043363d 8f39c501 00000000 00000000)

Module: "mark"
Event: "DCache miss": 35 samples
 #0  :  1 samples: [0x00001c70] mark.c:main(20:14)
        total latency=17       maximum latency=      17
        [0:7]=0    [8:15]=0    [16:31]=1  [32:99]=0  [100:inf]=0
 #1  :  5757 samples: [0x00001701] mark.c:main(23:21)
        total latency=   43132 maximum latency=     366
        [0:7]=4070 [8:15]=1668 [16:31]=18 [32:99]=0  [100:inf]=1
 #29 :  5786 samples: [0x00001700] mark.c:main(23:42)
        total latency=   40047 maximum latency=     439
        [0:7]=5294  [8:15]=433  [16:31]=55  [32:99]=0  [100:inf]=4
```

Figure 4: `pronto_tool` output

and analysis implementations. Current Pin instrumentation capabilities can extract only profiles that are not related to actual clock cycles. For example, taken branches, loop iterations counts, calls, memory references, etc.

Other examples of more sophisticated pintools-based technologies are helper threads [10.2] and hot stream data prefetch [2].

Helper thread technology, or software-based speculative pre-computation (SSP) was originated from complex database architectures and based on compiler-based pre-execution [10.1] to generate a thread that would prefetch long latency memory accesses in runtime. This is the 3-compilation model static technique. Its implementation is currently done in Intel Compilers [16] with Pronto mechanisms (option used for the first compilation

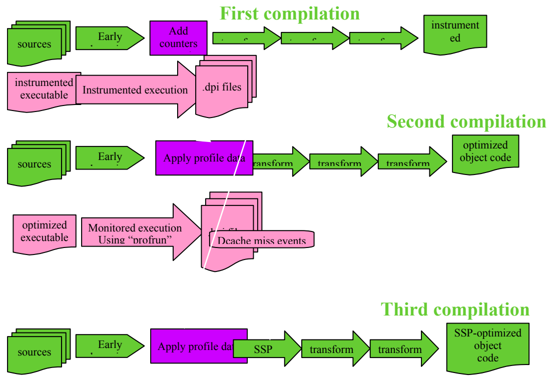

Figure 5: Helper Threads (SSP) build diagram

`--prof-gen-sampling`, for the second `--prof-use --ssp`, and for the third with `--ssp`). The workflow diagram is shown in Figure 5.

As a dynamic equivalent to this technique, the

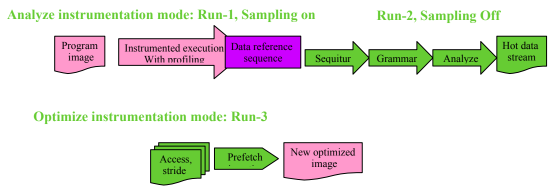

Figure 6: Hot data stream prefetch injection algorithm

mechanism described in [7] incorporates Pintool for dynamic instrumentation of memory read bursts in the sampling mode; the Sequitur for fast dynamic analysis of memory access sequences with long latencies; and matching mechanism for sequences that would benefit by prefetching (called hot data streams detection phase). Based on hot data streams, the Pin can inject prefetching instructions as shown in Figure 6.

6 Appendix B. Usage Model Examples On Proposed Workflows

Here are some potential applications of profile guided debugging and performance adaptive scheduling:

1. OS scheduling decisions may be based on occurring patterns of hardware performance events, event hazards detection or platform resource utilization hazards:

   Some event sequences can determine a hazard, upon which the OS scheduler may redefine priorities in the run queue and affinity to a logical/physical CPU.

2. Hyper-threading and Dual Core. Immediate performance gains. If a recurring pattern of utilization similar CPU resources was detected, the thread affinity assigned should distribute to run these threads on different physical cores. This approach expected to show immediate performance gains on HT-enabled system on a series of dedicated applications.

3. Independence of usage model while adopting Dual-Core/Multi-Core. In order to adopt DC/MC for maximizing the system performance, a user should be aware of system usage model. With performance adaptive scheduling infrastructure, the usage model alternation will become less relevant for performance. In turn, it may stimulate efficient adoption of multi-core technology by application developers, since user awareness of usage model will not affect extracting optimal performance from the software.

4. Simplify software development schemes. Background/foreground and process priority management based on performance
   .

5. Hybrid of OpenMP & MPI for high performance programming will be simplified. A performance-adaptive OS Scheduler will handle optimal scheduling dependent on processor's resource utilization for each OpenMP thread.

6. Power utilization optimization and energy control. Modern micro-architectures have an extensive set of energy control related performance counters. When power restrictions are enforced for a process execution, the number of stall cycles due to platform resource saturation should be minimized. The optimized scheduling for processes on preventing such platform performance hazards to occur should be handled by OS scheduler.

7. Dynamic Capacity planning analysis. Analyzing profiling data logs per thread and

detection of certain event sequence hazards may assist in identifying capacity requirements for the application.

8. Out-of-order execution layer for statically scheduled codes, better utilization of "free" CPU cycles and compensation for possible compiler's scheduler inefficiencies.

   Having performance feedback based OS scheduler will provide information with additional granularity (on top of the compiler scheduler) for filling the empty cycles generated by the compiler (or if present, even during OOO execution on x86).

9. Virtualization Technology. When a code is running on virtual processing units, and utilizing a virtual pool of resources, it is important to provide optimal performance, a dynamic code migration suggestion. The assignment between virtual and physical processing unit should be done based on actual performance execution statistics. If Linux is a "guest" OS, the presence of performance adaptive scheduling mechanism will allow the OS scheduler to be aware of resource utilization across all the virtual processes.

10. Profile-guided debugging proposal targets most difficult areas of debugging - performance debugging and scalability issues. See [10].6 on examples on how to utilize Helper Threads technology for memory debugging. By combining principles of Profile-Guided optimization and conventional debugging mechanisms we showed it is possible to architect a debugger's extension to set a breakpoint at a performance or power pattern occurrence. As a result, variety of metrics for the performance may be reflected in the debugging, such as: ratio mips/watt, instruction level parallelism. State-of-the-art debuggers allow users to manually define a breakpoint on expressions which involve values obtained from the memory during the application execution. This approach assists to extend the mechanisms to combine the expression values received from CPU and chipset performance counters in run-time.

    Examples of possible breakpoints which would be set by a user who debugs multi-threaded applications are:

    - Hazardous spin locks
    - Shared memory race conditions
    - Too long or too short object waits
    - Heavy ITLB, Instruction or Trace Cache misses
    - Power consuming blocks; Floating point intensive procedures
    - Loops with extremely low CPIs; low power blocks
    - Long latency memory bus operations
    - Irregular data structure accesses; alignment issues during run-time
    - Queue and pipeline flushes, unexpected long latency execution
    - Opcode or series of opcodes being executed
    - Hyper-threading contentions or race conditions
    - OpenMP issues

There are already working applications with Pin-based instrumentation for simulation and performance prediction purposes. Extending these simulation technologies [15], similar to the PGD technique generating *Interrupt 3*, we would be able to emit other interrupts, signals and eventually generate an alternate sequence of events.

References

[1] "Enhancements for Hyper-Threading Technology in the Operating Systems – Seeking the Optimal Scheduling", by Jun Nakajima and Venkatesh Pallipadi, Intel Corporation

[2] "Dynamic Hot Data Stream Prefetching for General-Purpose Programs", by Trishul M.Chilimbi and Martin Hirzel, Microsoft Research and University of Coloroado

[3] Compiler for PGO (profile feedback) and its repository data coverage: `http://gcc.gnu.org/onlinedocs/gccint/Profile-information.htm`

[4] Compiler optimizer, gcc4.1: `http://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Optimize-Options.html`

[5] Compiler for vectorization and reports: `http://www.gnu.org/software/gcc/projects/tree-ssa/vectorization.html`

[6] Pronto repository content, Profrun and pronto_tool – Intel Compiler tools, based on 2-compile model: `http://www.intel.com/software/products/compilers/clin/docs/main\_cls/index.htm`

[7] PAPI: `http://icl.cs.utk.edu/papi/overview/index.html`

[8] Pin & pintool: `http://rogue.colorado.edu/pin`; Pin manual for x86: `http://rogue.colorado.edu/pin/documentation.php`; Pin related papers: `http://rogue.colorado.edu/pin/papers.html`

[9] **Sequitur: For Sequitur Algorithm description see**:

[9.1] "Compression and explanation in hierarchical grammars", by Craig G. Nevill-Manning and Ian H. Witten, University of Waikato, New Zealand

[9.2] "Identifying Hierarchical Structure in Sequences: A linear time algorithm", Craig G.Nevill-Manning and Ian H.Witten, University of Waikato, New Zealand, 1997

[9.3] "Efficient Representation and Abstractions for Quantifying and Exploiting Data Reference Locality", by Trishul M.Chilimbi, Microsoft Research, 2001

[10] **Helper threads and compiler based pre-execution:**

[10.1] For concept overview see "Compiler Based Pre-execution", Dongkeun Kim dissertation, University of Maryland, 2004,

[10.2] Threads: Basic Theory and Libraries: `http://www.cs.cf.ac.uk/Dave/C/node29.html`

[10.3] Usage model for helper threads in "Helper threads via Multi-threading", IEEE Micro, 11/2004

[10.4] "Helper Threads via Virtual Multithreading on an experimental Itanium 2 Processor-based platform", by Perry Wang *et al*, Intel, 2002

Helper threads and pre-execution technology for:

[10.5] Profiling, see "Profiling with Helper threads", T.Tokunaga and T. Sato (Japan), 2006

[10.6] Debugging, see “HeapMon: A helper thread approach to programmable, automatic, and low overhead memory bug detection”, IBM Journal of Research and Development, by R.Shetty et al., 2005

[11] “Dynamic run-time architecture technique for enabling continuous optimizations” by Tipp Moseley, Daniel A. Connors, etc., University of Colorado

[12] “Chip Multithreading Systems Need a New Operating System Scheduler” by Alexandra Fedorova, Christopher Small, et all, Harward University & Sun Micro.

[13] “Methods for Modeling Resource Contention on Simultaneous Multithreading Processors” by Tipp Moseley, Daniel A. Connors, University of Colorado

[14] “Pthreads Primer, A guide to multithreaded programming”, Bill Lewis and Daniel J. Berg, SunSoft Press, 1996

[15] SimPoint toolkit by UCSD: `http://www-cse.ucsd.edu/~calder/simpoint/simpoint_overview.htm`

[16] Intel Compiler Documentation – keywords: Software-based Speculative Precomputation (SSP); Profrun utility, prof-gen-sampling: `http://www.intel.com/software/products/compilers/clin/docs/main_cls/index.htm`

[17] “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,” by Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, Kim Hazelwood. Programming Language Design and Implementation (PLDI), Chicago, IL, June 2005

[18] Tree SSA: A new optimization infrastructure for GCC, by Diego Novillo, Red Hat Canada, 2003: `http://people.redhat.com/dnovillo/pub/tree-ssa/papers/tree-ssa-gccs03.pdf`; `http://gcc.gnu.org/projects/tree-ssa/#intro`