Final report Dagstuhl Seminar on Emerging Uses and Paradigms for Dynamic Binary Translation

Erik Altman, IBM TJ Watson Research Center
Bruce R. Childers, University of Pittsburgh
Robert Cohn, Intel - Hudson
Jack Davidson, University of Virginia
Koen De Bosschere, Gent University
Bjorn De Sutter, Gent University
M. Anton Ertl, TU Wien
Michael Franz, Univ. California - Irvine
Yuan Gu, Cloakware/Irdeto - Ottawa
Matthias Hauswirth, University of Lugano
Thomas Heinz, Robert Bosch GmbH - Stuttgart
Wei-Chung Hsu, University of Minnesota
Jens Knoop, TU Wien
Andreas Krall, TU Wien
Naveen Kumar, VM Ware, Inc.
Jonas Maebe, Gent University
Robert Muth, Google - New York
Xavier Rival, ENS - Paris
Erven Rohou, INRIA - Rennes
Roni Rosner, Intel Israel
Mary Lou Soffa, University of Virginia
Jens Troeger, Microsoft Research - Redmond
Christopher Vick, Sun Microsystems - Menlo Park

26 October 2008 – 31 October 2008

Dagstuhl Seminar Proceedings 08441
Emerging Uses and Paradigms for Dynamic Binary Translation
http://drops.dagstuhl.de/opus/volltexte/2009/1888

1 Introduction

Software designers and developers face many problems in designing, building, deploying, and maintaining cutting-edge software applications–reliability, security, performance, power, legacy code, use of multi-core platforms, and maintenance are just a few of the issues that must be considered. Many of these issues are fundamental parts of the grand challenges in computer science such as reliability and security.

As an example, consider reliability. No serious software expert would claim that the current state-of-the-art in software engineering allows the construction of even moderately complex software that does not contain flaws or bugs. Perhaps a more realistic approach is to acknowledge that, no matter what process is used (i.e., rigorous design practices, thorough software testing, etc.), the delivered software will inevitably contain imperfections. If we accept this fact, it is then expedient to consider approaches that address these flaws after delivery. Certainly a natural approach would be to consider how to build software that adapts to flaws as it runs.

Similarly, consider the hardware evolution. As technology scaling continues, hardware devices exhibit more variability in their crucial performance metrics. There is both variability over multiple devices that leave the fab, and variability within single devices over their lifetime. Some components can behave erratically or completely fail. Software must be able to adapt to this changing computational substrate.

The underlying theme is that modern software must be able to adapt–whether to an existing bug, a security attack, changing power availability, or a failing core. No longer do we build software with the notion of constancy, rather we fully embrace dynamism and develop the theories and capabilities that allow software to adapt dynamically. The resulting software is more robust, can adapt to a changing computational environment, and is potentially cheaper and faster to build.

Dynamic binary translation (DBT) has gained much attention as a powerful technique for the run-time adaptation of software. It offers unprecedented flexibility in the control and modification of a program during its execution. Some of the uses of DBT include emulating and simulating instruction sets, monitoring and optimizing performance at run-time, providing resource protection and management, virtualizing resources, and detecting malware. In many cases, virtualization looks like a very promising paradigm, and in such cases DBT is the primary means for achieving and implementing virtualization.

In the context of this report, DBT refers to any run-time code manipulation that involves transforming lower-level code (either precompiled or manually written). This includes, but is not limited to, (1) just-in-time compilation of bytecode into native instruction sets (ISAs or instruction set architectures), (2) run-time transformations of native code for the purposes of optimization, instrumention, adaptation, etc., and (3) translation of code for one ISA into code for another ISA. It includes such transformations that are applied both at the process level and at the system (e.g., OS or hypervisor) level.

While dynamic binary translation has been used to solve many problems, there are newly emerging software requirements, including self healing, security, reliability and constraint awareness, where DBT can play a role. For software adaptivity and self healing, DBT provides the ideal vehicle to support dynamic software changes and patches that fix errors, introduce/remove software functionality, or change software versions based on run-time constraints. For security, DBT can inspect application code to check for malicious behaviour, modify the code to increase its resilience to outside attack, and obfuscate the code to hinder reverse engineering and to protect intellectual property. In the case of

reliability, DBT can inspect/modify an application to check for error cases that are not anticipated, validate dynamic component usage contracts, and modify the program during execution to avoid failed hardware units or software components. Lastly, for multiple constraints, DBT can make trade-offs with run-time information and between constraints, such as performance, energy consumption and memory usage, that are unknown a priori to program execution or that change during execution.

At the same time that these new requirements are being imposed, paradigm shifts are happening in the underlying hardware architecture that affect DBT. One change involves embedded systems. Whereas many embedded systems used to be very rigid hardware implementations for specific applications, current systems are using more powerful and more flexible hardware and software capabilities in the form of system-on-a-chip designs with multiple heterogeneous cores. Furthermore, embedded systems are increasingly subject to the same requirements as general-purpose computer systems, such as operating in networked environments and upgradability. In some cases, the requirements are even more critical and stringent: secure and reliable software is quite important in this domain. Self healing and adaptive software are particularly desirable in embedded environments where the system itself cannot be easily swapped for a new one (e.g., a satellite or planet explorer). To deal with these additional requirement in software-only or hardware-only solutions seems very difficult, if only because of the unacceptable overhead that such solutions would introduce. Instead, cross-layer solutions are required. Through virtualization, DBT can provide the necessary support for such cross-layer solutions without limiting the portability of software, and without constraining the hardware to backward compatibility requirements.

While DBT offers many benefits, there are, however, also unique challenges that have so far limited DBT's use in embedded devices. Embedded systems have tight memory and performance demands, while DBT has typically been used in systems with relatively large memories and fast processors. Also, embedded systems have strict timing and verification/validation requirements; for example, hard real-time systems use static timing analysis to ensure that application code is guaranteed to meet deadlines. Because DBT translates a program as it executes, it introduces uncertainty into the timing performance of the application. Furthermore, run-time code changes make static software verification impossible. For DBT to play a role in achieving software requirements for embedded systems, solutions to these challenges must be found.

The other hardware paradigm shift involves multi-core processors: computer architectures and software applications are quickly moving toward large-scale parallel organizations (including heterogeneous ones) where tens or perhaps even hundreds of cores are on a single processor chip. The software application and operating system can enlist the aid of multiple cores in tackling an application problem. To date, DBT has largely focused on single-core systems. The introduction of multi-core systems demands that DBT provide inherent support for parallel execution and to exploit the parallelism in the underlying hardware substrate.

This Dagstuhl Seminar brought together experts from research and industry to discuss and identify the problems and promising directions for the new software requirements and hardware architectures described above. During our seminar, we identified significant challenges and opportunities in six areas of DBT research.

1. An open infrastructure for DBT

2. DBT and parallelism

3. DBT and hardware variability

4. DBT and real-time requirements

5. DBT and validated/certified software

6. DBT and security and protection

These are summarised in the following sections.

2 Challenges and opportunities

2.1 An open infrastructure for DBT

The pervasive use of a DBT system to meet the demands placed on modern software applications is a radical departure from existing software development methodologies that largely view software as a static entity that does not change during execution. Consequently, in addition to the research necessary to achieve software frameworks that support a control-system view of software, there is also much research necessary to explore the potential ramifications of such an approach on software development practices.

We propose a research thrust that focuses on investigating the use of DBT as a control system for software. This notion of using DBT as a control system for software offers the potential for "game-changing" software and hardware development paradigms.

Figure 1 shows the classic capabilities of a DBT control-system.

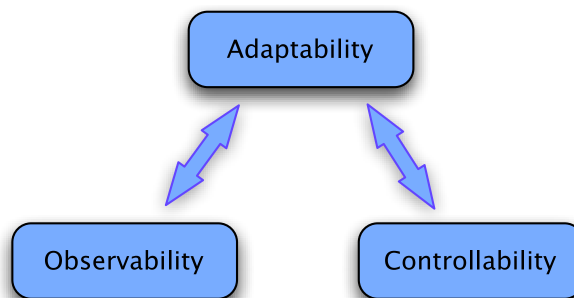

Figure 1: Control system capabilities

Observability is the ability for the DBT system to observe and collect data about the running application. This capability is achieved through *sensors* that the control system places (and removes) in the application to collect the necessary information so the system can adapt to achieve some goal. In the case of performance (which has been well studied), methodologies have been developed for measuring cycles executed, cache misses, page faults, etc. For other DBT control systems, development of new hardware/software sensors is needed. For example, how does the system detect that a cache line is failing, or that an adversary is attacking an application?

Controllability is the ability for the DBT system to control or effect changes in the running application. This capability is achieved through *actuators* that the control system uses to adapt the running application. These actuators may be built into

the application or inserted by the control system dynamically. In the case of performance, methodologies have been developed for selecting what code to execute. For other DBT control systems, new actuators will be required. For example, how does the system recover data that has been corrupted by a sensed attack?

Adaptability is the ability for the DBT system to make decisions based on the data provided by the sensors and to invoke the appropriate actuators. For adaptation, the system must continually monitor its success in achieving the stated goal. When the system is not achieving its goal, it must use actuators to modify itself so that it moves into a state where the goal can be satisfied. For example, in a multi-core system where thermal constraints dictate that a core (or cores) must be released, the DBT system must determine the new mapping of tasks to avoid a thermal emergency yet still achieve acceptable performance.

A key challenge of this research is the development of appropriate domain-specific languages for expressing the mechanisms that a DBT control system should use to observe and control an application as well as expressing high-level constraints and contracts that should be met and enforced.

2.2 DBT and parallelism

Hardware systems increasingly incorporate parallel components, creating opportunities beyond classical instruction-level-parallelism (ILP). These opportunities include data-level parallelism (DLP) in the form of vector data types and SIMD instructions; instruction-level parallelism in the form multi-core (MC) and multi-threading (TLP) and combined data; as well as combined data and control level parallelism in the form of single-program multiple-data (SPMD) models. Parallel software, on the other hand, is difficult to write, debug, validate and maintain. It evolves mainly in specialized areas such as databases, graphics and some scientific applications.

On the one hand, DBT systems will be challenged with the difficult task of extracting parallelism from either legacy or new code. This extraction might be automatic or semi-automatic - aided by hints and directives from higher layers - user, compiler or operating-system. The extracted parallelism could be fine or coarse-grain DLP, TLP or vectorization. It may be also combined with speculative modes of execution such as data-speculation or run-ahead threads. Yet another contribution of a DBT system could be the scheduling of parallel tasks. Dynamic information might be used for improved parallel-resource utilization by means of smarter and adaptable scheduling mechanisms at different levels, such as Java virtual machines (JVM), OS or virtual machine management (VMM).

On the other hand, the DBT approach can be integrated into new models of programming. These new models support more effective parallel programming by providing better memory abstraction such as transactional memory (TM), alternative approaches to program resilience (n-version programming) or data-sharing (SPMD).

There are many challenges facing a DBT system expected to perform correctly and efficiently in a parallel environment.

DBT systems need to be retuned from sequential to parallel oriented environment. This applies to all aspects of DBT, including hot-region (or trace) selection and formation, dedicated memory (e.g. code-cache) management, choice of optimization, performance measurement and feedback mechanisms.

A major effect of parallelism is to significantly increase the complexity of the environment in which the DBT system operates, thus magnifying whatever challenges it already faced in a less parallel setting: The increased number of race conditions, external events, coherency traffic and alike requires careful design of the DBT system to avoid deadlock or live-lock situations and circumvent unbearable performance drops. Complex execution models, such as open-nesting in TM, are very difficult to handle.

Verifying the correctness of a DBT system is a significantly harder task in a parallel setting. In particular, validating optimization and instrumentation of multi-threaded programs is hard and would require newer validation techniques, on top of those considered for sequential programs.

It is therefore of great importance to assist in the adoption and exploitation of parallel hardware and parallel models of computing in the context of dynamic binary translation (DBT).

2.3 DBT and hardware variability

The term “hardware variability” can refer to many things:

Heterogeneous multi-cores In the embedded world, multi-processor-on-a-chip designs feature different kinds of cores. Typically, there is a control core that runs an (RT)OS, and a number of DSPs or accelerators. Likewise, general-purpose multi-core processors may feature different cores. These can be different implementations of the same ISA, or cores that feature different ISAs. Life migration between cores of a heterogeneous multi-cores will require that the software adapts itself to varying hardware characteristics.

Heterogeneous clouds Server farms, also called computing clouds, may contain different kinds of multi-core processors. When processes are able to migrate dynamically between different processors, this variability can be seen as variability over time.

Reconfigurable cores FPGAs and other reconfigurable components can be considered as processing elements that can vary over time. Similarly soft-cores can be implemented implemented on top of FPGAs.

Process variability Process variability can also be treated as processor variability, as processors with varying properties will leave the fab. The differences in properties may be observed in terms of clock frequency, power consumption, other performance characteristics, or of functionality, or of defects. Furthermore, process variability can be seen as hardware variability over time, as defects will occur as a result of too much stress or wearout.

In all of these cases, a programmer ideally should be able to write software in a processor-independent manner. This approach ensures that the software can run on all potential target processors, and potentially that a running program can migrate between different cores. We believe that total virtualization is the best paradigm to achieve this. To support this paradigm, DBT is the most promising tool. As such, the above types of hardware variability open up many opportunities for DBT.

When a process migrates between different cores, for example between same-ISA cores with different cache sizes, the process should be re-optimized for its changing environment. For example, upon migration a loop nest might be rewritten to match the working set

of the inner loop to the new cache size. A scenario to support this adaptation consists of hardware-independent source code, a compiler that generates meta-information about the rewriting (and parallelization) options of the loop, and a DBT that can interpret the meta-information to perform the actual rewriting. Fundamentally, the meta information describes user scenario parameters such that the static compiler does not need to generate separate code for each scenario.

One of the shortcomings of some existing virtualization systems is that they inform the guest OS of the underlying hardware. For example, when an OS is installed inside a VM running on an x86 core, the VM sometimes informs the guest OS about the presence of certain SSE ISA extensions. From then on, that guest OS assumes the presence of those extensions, which prohibits migration to x86 cores that lack them. To overcome this problem while still being able to exploit all available hardware functionality, fully hardware-independent guests should be supported by means of DBT VMs. In many cases, abstract meta-information about the available hardware and about the required resources will be required to enable the exploitation of hardware features for software that was written in a hardware-independent manner.

With respect to process variability issues and wearout, workarounds for processor defects, processor inefficiencies or emerging defects (due to stress) will be much easier to implement by means of DBT, as many of the wearout issues occur on very small time scales. In the case of transistor overstressing or temperature hot spots, the timescales are often not longer than hundreds of microseconds.

The ability to adapt or replace soft cores at run-time in the context of reconfigurable hardware, together with the adaption of the running software via DBT, can open up new opportunities. Similarly, with DBT long-running client-side simulations could continue executing when hardware is replaced, without requiring intrusive or expensive checkpointing support. Likewise, with DBT, fixed cores of which components can be disabled or enabled on the fly could be used to adapt a system to changes in its surroundings. For example, during the day when the energy cost peaks, power-hungry components could be disabled. With the support of DBT, the running software can then simply be adapted on the fly.

A final area of emerging opportunities in the context of hardware variability and full virtualization is the area of fault tolerance. Full virtualization by means of DBT enables duplicating the execution of a program on different hardware instances. Compared to duplicating the execution on multiple instances of the same hardware, this would reduce the risk that identical faults occur in the instances. With the current state of the art, different (types of) applications may require different (types of) hardware to run on. To support fault tolerance, all this hardware needs to be duplicated. It seems likely that when all software would be able to run on identical hardware, of which it is shielded by means of virtualization and DBT, less hardware duplication would be required to provide the same level of fault tolerance.

We envision two major challenges to exploit the above opportunities.

First, much meta-information is required to describe resource requirements of applications (for example, required QoS of a network-on-chip to deliver predictable RT performance), to describe available hardware resources, to provide HW feedback (via virtual performance counters) and to describe optimization and parallelization options of code to run. Determining the right abstractions for this meta-information is an open challenge.

Secondly, run-time resource management by means of a DBT can be done at a much finer granularity than existing (design-time) methodologies. We know of no existing work

in this field, so this is a very open challenge.

2.4 DBT and real-time requirements

Today's technology is more and more controlled by and dependent on real-time systems. This dependence makes real-time applications of fast growing importance both for the continuing technological progress and social welfare of our society as a whole, and for the safety and convenience of the daily live of all its individuals. In contrast to many other application fields of practical relevance, dynamic binary translation has not yet successfully penetrated this field.

One of the reasons for this lack of use might be the fact that the construction of systems satisfying hard real-time constraints and proving that these systems meet their deadlines is a challenge even still for statically compiled systems, far from being a routine engineering exercise. Dynamic compilation, as in JIT dynamic binary translation, exacerbates these problems. The fact that the worst-case execution time (WCET) of a program (part) can be very different from its average execution time prevents the provision of timing guarantees for dynamically compiled code from being an easy endeavour. We thus conjecture that a JIT dynamic binary translation approach towards a RT-DBT will probably not work in the foreseeable future.

Approaches based on ahead-of-time translation, especially boot time translation or static (pre-)translation, are more controllable by the system developer, and hence more likely to be successful in the short to mid-term. We think that it is essential to the breakthrough of such approaches that code discovery and WCET analysis are performed statically and that their findings are passed as meta-data to the RT-DBT. By contrast, we conjecture that in following such an approach RT-scheduling will probably best be a task of the RT-DBT.

In conclusion, we note that we consider the successful virtualization of RT-devices is of particular importance and practical relevance for the further advancement of the field, and hence a demanding challenge for the DBT community.

2.5 DBT and validated/certified software

Dynamic binary translation systems are inherently complex, hence, therefore they should be validated before being used in critical applications. For instance, the use in web-browsers may lead to security risks when personal data is sent by a dynamically translated application (an incorrect translation could cause more private data be sent than intended). Similarly, software used in embedded systems should meet stringent safety criteria (a mistranslation could cause life-threatening accidents).

We believe that the validation of dynamic translation techniques should be a first move towards their certification, for use in specific and demanding areas such as embedded systems. Certification of such techniques is definitely a major, long term challenge. Validation should be easier to tackle first.

The first step for validation is to specify which program transformations we should validate. Validation techniques require a clean definition of the semantics. Hence, we should model families of dynamic translation techniques. In particular, what is preserved by transformations should be taken into account here. Several classes of transformations may need be defined: for instance, translation invariants should be much stronger in the case of a mere instruction replacement transformation than in a global optimization one,

with on-stack replacement of dynamically optimized function code.

The second step to achieve validation is to assess existing validation techniques (e.g., those used for static compilation) and to derive new techniques. We should not expect abstract invariant translation (aka, proof-carrying code) approaches to work on legacy systems, since these require a significant amount of metadata to be part of the executable in the first place.

Translation validation techniques (i.e., per translation, automatic proof of semantic equivalence) look however more promising. In particular, we could imagine deriving a dynamic equivalent from the classical translation validation schemes. In this case the proof of equivalence would be performed at run-time, right after dynamic translation but before the execution of the translated code. However, in the case where the proof of equivalence fails a fall-back mechanism should be provided, so as to continue the execution (this could be done by a simpler, validated interpreter). It should also be noted that the validation should not become a run-time bottleneck, hence efficiency is critical.

We could also consider the validation of the transformation itself. This validation would be done statically, e.g., by formally proving the soundness of the transformation using a proof assistant. Such a proof would be carried out once in the lifetime of the dynamic translation tool. Recent successful projects have shown this to be doable for static compilers. However, it is unclear whether this approach is still feasible in the face of the increased complexity of dynamic translation tools.

Testing has also been used to validate static compilers (e.g., Ada compilers). This approach could also contribute to the validation of dynamic translation tools. In order to achieve this goal, a good coverage of all execution cases should be met.

Finally, we note that dynamic binary translation is also an opportunity for new research in trusted compilation/code transformation. Indeed, validation could be integrated into the design of new binary translation frameworks. For instance, we could imagine defining metadata formats to include in binaries that will later be dynamically translated. This approach could have several significant advantages over existing techniques. It could be the basis for cross-platform, proof-carrying code systems. Metadata could also contain hints for the run-time provers used in dynamic translation validation systems. When the binary translation is performed in several stages (for instance, when a part is carried out statically and the rest is performed at run-time), metadata could also contain certificates for correctness of the parts translated in the first phase; this would allow speeding up later verification stages.

2.6 DBT and security and protection

DBT can offer capabilities to assist with improving software security and protection that cannot be easily achieved using other mechanisms. For this to be possible, DBT however first has to meet certain requirements and overcome a number of challenges.

Language and platform (in)dependent security features DBT as part of an executable run-time environment can provide a unique opportunity to facilitate both platform-dependent and platform-independent security features, as well as language-dependent features. At the same time, it should not restrict the use of techniques employed by general purpose (static) software hardening tools;

Tunability of security and performance In general, software security and performance are conflicting goals trade-off. For some applications, a moderate level of

security is good enough and performance needs to retain a certain minimum level, e.g., a secure application run on an embedded systems with restricted resources. DBT should provide a way to trade off performance and security to meet application requirements for both. The adaptability feature of future DBT infrastructures could be used for this purpose.

Security functionality plugins DBT should provide a means to integrate prebuilt security plugins. There are a number of reasons for such functionality:

- Such functionality can provide a new mechanism for execution security providers to take advantage of dynamic security capabilities such as dynamic transformations and diversity that can be very effective to prevent dynamic attacks;
- Ease of upgradability and downgradability of execution security and protection offerings;
- Enable an application or the target system to apply specific security measures;
- DBT security itself can be easily managed.

Of course, such mechanism must not enable circumvention of these same hardening measures.

Security metadata Security should be specified and checked between many levels and at different granularities. Users can annotate their source code and source level protection tools should generate security metadata for DBT using this information. Binary-level protection tools then enable the use of this metadata, propagate it and generate security metadata for DBT. DBT also enables handling security metadata from other sources. All this metadata should be represented in a open and uniform form so it can be accepted, interpreted and applied dynamically by any DBT;

Robustness If DBT is used for security purpose, the robustness of DBT is of paramount importance. Therefore, DBT should distinguish between internal development mode and external deployment mode:

- Development mode: open, transparant, debugable;
- Deployment mode: closed, not observable, not debugable.

3 Conclusion

This reports describes 6 promising areas of opportunities and challenges in which DBT research could lead to breakthroughs with a major impact on system design and deployment:

1. An open infrastructure for DBT that is observable, controllable and adaptable is essential to advance DBT research and to implement various techniques.
2. If DBT wants to take advantages of modern parallel platforms, it will have to learn how to run parallel code (including (re-)parallelizing code).
3. Effectively dealing with hardware variability might be one of the future killer applications of DBT, both at the component and at the core level.

4. Being able to guarantee real-time requirements in a DBT will open up a whole new application domain (real-time embedded systems).

5. Being able to validate or certify software will also open a complete new application domain (safety critical applications).

6. Modern software has ever increasing security and protection requirements. DBT can help in providing an additional (dynamic) security layer, possibly as a seperate concern (i.e., not hard coded in the application).