DARCO: Infrastructure for Research on HW/SW co-designed Virtual Machines

Demos Pavlou[‡,1], Aleksandar Brankovic, Rakesh Kumar, Maria Gregori
[†]Universitat Politécnica de Catalunya
{abrankov, rkumar, mgregori}@ac.upc.edu

Kyriakos Stavrou, Enric Gibert, Antonio Gonzalez[†,‡]
[‡]Intel Barcelona Research Center (IBRC) – Intel Labs
{demos.pavlou, kyriakos.stavrou, enric.gibert.codina, antonio.gonzalez}@intel.com

ABSTRACT

One of the major problems of academic research is the unavailability of appropriate tools where researchers can develop and evaluate their ideas. Research groups tend to spend a significant amount of time in developing tools which results in an abundance of incomplete tools which do not provide sufficient features for other groups to use. Often these tools are never made public which makes the reproducibility of the results difficult and time consuming.

This paper presents *DARCO*, an enabler infrastructure for research in the field of HW/SW co-designed virtual machines. *DARCO* models a HW/SW co-designed system with different guest and host ISAs. Its Emulation Software Layer (ESL) translates and optimizes x86 binaries to run on a PowerPC processor. The ESL provides staged compilation including an interpreter, a translator, and an optimizer. *In addition to the functional models, DARCO provides timing simulators and a powerful debugging toolchain.* *DARCO* has a clean interface for including new optimizations in the ESL and allows easy implementation of new hardware features. *DARCO* has a functional emulation speed of 8 million x86 instructions per second and timing evaluation speed of 400k x86 instructions per second.

Keywords

Co-designed virtual machines, dynamic binary translation

1. INTRODUCTION

Hardware/Software co-designed processors gained a great momentum during last years. These architectures have important advantages over traditional hardware-only systems like the exploitation of dynamic information, the ability to provide Front-Ends of different ISAs and the segment-specific optimizations. Recently academia and industry focused on software layers running on top of off-the-shelf processors with the Java runtime [14] and managed systems like the Microsoft .NET being the main examples. Efforts based on HW/SW co-design however have been sporadic and the field has not yet reached a critical mass.

Although Transmeta delivered two products based on the HW/SW co-design paradigm, Crusoe [1,4] and Efficeon [8], academia has not invested much effort on such architectures. The few projects that we are aware of, like DAISY and BOA [6,11], are limited to either studying specific components of the HW/SW co-design or to adding minimal SW support to existing HW systems. We strongly believe that the reason academia has not dedicated a critical mass to this paradigm is not the lack of trust into the technology but the lack of tools.

Such a toolchain needs to provide functional emulators of the host and guest ISAs (in case they are not the same), cycle-accurate timing simulation for the host processor and a software layer that is able to interpret, translate and dynamically optimize the guest binaries. Developing from scratch and debugging all these components has a multiple man-years cost. Using existing components to build such an infrastructure is an alternative. The need to deeply understand, adapt and glue these components together would however lead to a similar cost.

This paper presents DARCO, an effort of multiple man-years that led to a powerful infrastructure for research on the HW/SW co-designed paradigm. To enable a broader research spectrum DARCO has a different host and guest ISA. In particular, it provides a full-system x86 guest Front-End that is translated, optimized and executed on a Power PC (PPC) processor. The key components of DARCO are the x86 and PPC functional models, the Emulation Software Layer (ESL), the timing simulators and the accompanying debugging and monitoring tools. Except for the functional models for which we used heavily modified versions of QEMU [10], all other components are in-house developments.

DARCO is not an early version of an envisioned infrastructure but a mature and debugged tool. It has a pass rate of 100% for all the SPEC CPU 2006 [13] benchmarks. Its SW layer, the ESL, is equipped with an interpreter, a translator, a profiler and a dynamic optimizer that applies a plethora of optimizations to the translated regions. DARCO is a re-

1. The presented work was completed while Demos Pavlou was a Research assistant at Universitat Politecnica de Catalunya.

search enabler that allows exploration of all the components of a HW/SW co-designed architecture.

In this paper, we do not limit the discussion to presenting the infrastructure but we also characterize the ESL using SPEC CPU 2006. In section 2 we present DARCO and the ESL. In section 3 we present statistics regarding the speed of DARCO and characterization of ESL and SPEC CPU 2006 applications. Some related work is discussed in section 4 and our concluding remarks are presented in section 5.

2. DARCO INFRASTRUCTURE

DARCO is an infrastructure for research on co-designed virtual machines. It emulates a co-designed processor which is executing x86 applications on a PPC processor through dynamic binary translation and optimization.

DARCO provides an Emulation Software Layer (ESL) which implements the layer of abstraction between the target x86 ISA and the host PPC hardware. The reasoning behind selecting x86 as the target ISA is its wide usage. As for the selection of PPC as the host ISA, it is also based on its wide usage and its support for vector instructions on which we can map several of the x86 SSE instructions.

The ESL is implemented in a modular way in order to simplify the addition of new features, e.g. new optimization techniques. *DARCO* is also designed in a way that eases the study of new hardware features and host ISA extensions.

The current version of *DARCO* models only user-level x86 instructions. Future versions will support full-system emulation.

2.1 Main components

DARCO consists of four main components which interact as depicted by Figure 1. These are: the *x86 component*, the *PPC component*, the *Timing simulator* and the *Controller*.

The ***x86 component*** provides an x86 full-system functional emulator on top of which an unmodified operating system is executing. The *x86 component* keeps the correct architectural state and is the only component that interacts with the Operating System.

The ***PPC component*** provides the processor functional model for DARCO. The PPC functional emulator is executing the ESL which translates and optimizes the x86 instruction stream to PPC instructions. The PPC component has been updated to support additional instructions used by the ESL as described in section 2.3

The ***timing simulator*** models a parameterized PPC in-order core. It receives the dynamic instruction stream from the PPC component and provides detailed execution statistics. The use of the timing simulator is optional and doesn't affect the functionality of the rest of the components.

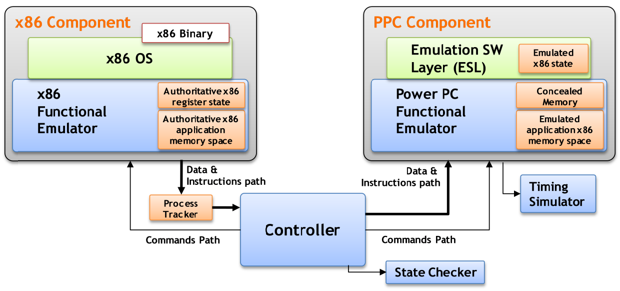

Figure 1: DARCO main components.

The ***Controller*** is the main interface of DARCO with the user. It provides full control over the execution of the application as well as debugging utilities. The main task of the controller is the synchronization of the execution of the other components and the resolution of the various requests from the PPC component (Section 2.2).

The x86 and PPC functional emulators are heavily modified versions of QEMU (Quick EMUlation tool) [10]. The rationale behind selecting QEMU as functional emulator is that it is a constantly updated and improving tool. This allows DARCO to benefit from improvements done over time. Moreover, the high execution speed of QEMU is a significant factor, since it helps the infrastructure to be more efficient. As for the rest of the components, i.e. the ESL, the controller and the timing simulator they are in-house developments.

2.2 Execution Flow

The execution flow of an application passes through three distinct phases; *initialization*, *execution* and *synchronization*. During the *initialization* phase, the controller first starts the PPC component which in turn, initiates the execution of the ESL. The PPC component then remains idle until the Controller sends the x86 register state of the application to be executed to it.

As for the x86 component, when launched, it initiates the execution of the application defined by the user. When it reaches the system call EXECVE (which always takes place at the beginning of an application) the execution pauses. A process tracker is initialized with the application's CR3 value, which can be used to distinguish the specific process from the rest of the applications running on top of the operating system. The process tracker is used throughout the execution of the application in the x86 component in order to ease synchronization and tracking of the changes made to the x86 state, register and memory of the application. After the process tracker is initialized, the x86 component sends the initial x86 register state of the application to the Controller. The initialization phase is completed when the Controller sends this state to the PPC component. At this point in time, the x86 register state is the same in both components.

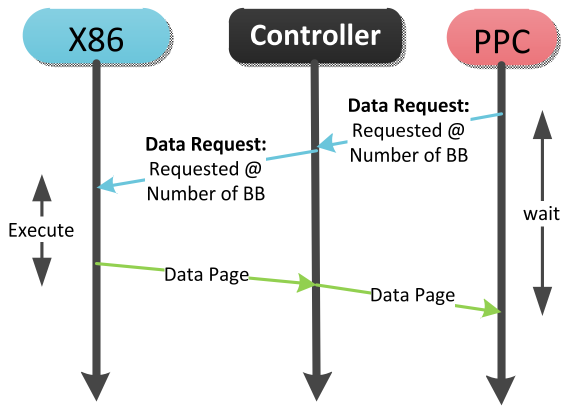

Figure 2: Data page request from the PPC component, enforcing the synchronization phase.

During the *execution* phase, the ESL begins by executing code from the initial `%eip` it received during the initialization phase. All changes made to the x86 register state from the emulation of the x86 instructions are stored in the "*Emulated x86 register state*" which resides in the memory space of the ESL. Changes made to the memory space of the x86 application are stored in the "*Emulated application x86 memory space*" which also resides in the memory space of the ESL. While the x86 application is making forward progress under the ESL, the x86 component remains idle and its memory state untouched.

The *synchronization* phase is initiated by the PPC component when any of following three events occur during the execution phase; (1) *data request*, (2) *system call* or (3) *end of application*. The data request event is raised when the PPC component encounters a load or store instruction that accesses an x86 memory location for the first time. The subsequent actions from the other different components are depicted in Figure 2. The PPC component sends a request to the Controller for the particular data page along with the total number of dynamic x86 basic blocks that were executed until this point. Then, it remains idle until the request is satisfied. The Controller forwards the request to the x86 component, which in turn continues the execution of the application until it reaches the same execution point as the PPC component (remember that the x86 component remained idle after the initial launch of the application). When the correct execution point is reached, the data page is sent to the Controller and forwarded to the PPC component. This process guarantees that after every synchronization phase, the x86 application state, register and memory, is identical between the x86 component and the ESL. Otherwise the system complains and execution is aborted. This is also a useful technique to debug DARCO. The exact same process is followed for the other two events, *system calls* and *end of application*.

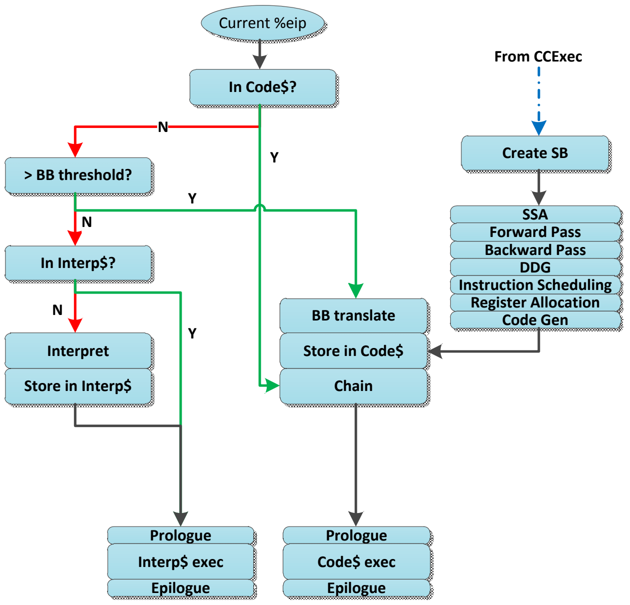

Figure 3: Emulation Software Layer execution flow. The left path is followed in IM, the middle in BBM and the right in SBM.

System calls raise the synchronization event because the ESL only models user-level code. The synchronization phase will fetch the modifications done by a system call from the x86 component. As for the end-of-application, the synchronization phase is necessary in order to verify that the execution of the application on the PPC component was correct.

2.3 Emulation Software Layer (ESL)

The ESL is the software layer that executes on-top of the PPC processor. It is responsible for translating the target x86 code to the host PPC ISA. In a nutshell, the ESL has four different execution modes; *interpretation* mode (**IM**), *basic block translation* mode (**BBM**), *superblock and optimization* mode (**SBM**) and *code cache execution* mode (**CCExec**).

The ESL starts by interpreting the x86 instructions. When a basic block (BB) executes more times than a predefined threshold the ESL switches to BBM for this BB which is translated and stored in the code cache. The subsequent execution of this BB is done in CCExec and profiling information is gathered regarding the direction of the branch and its target. When a BB reaches another repetition threshold, it triggers SBM. During this mode, the control flow profiling information that was collected during the CCExec mode is used by the translator in order to create a superblock (SB) with starting point the BB that triggered SBM. The SB passes through several optimizations (Section 2.3.2) and is stored in the code cache. Subsequent executions of the SB are done in CCExec mode. The high level view of the execution flow of ESL is shown in Figure 3.

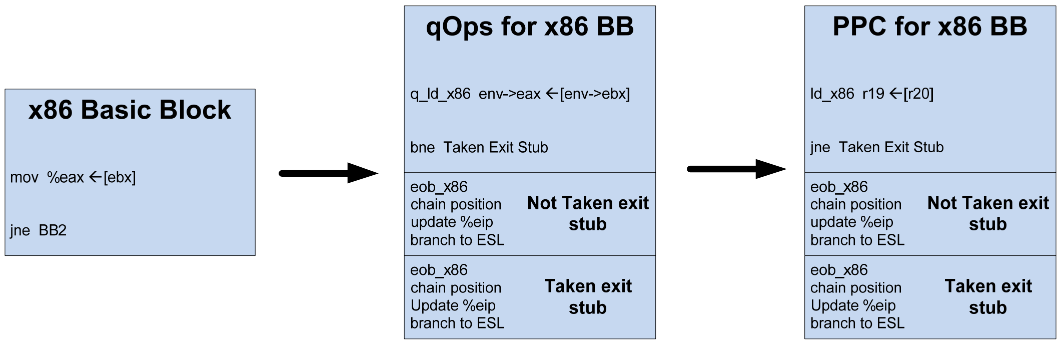

Figure 4: Abstract translation of an x86 BB to PPC. The eob_x86 instruction is used by DARCO for execution synchronization and special ld_x86 instructions to point out accesses to x86 memory space.

In the following sections we provide a more detailed description of the different modes of ESL but due to space limitations we only discuss the most important scenarios.

2.3.1 Interpretation and basic block translation

The ESL begins the execution of the application in IM. While in IM mode, x86 instructions are interpreted one by one and the x86 state is updated accordingly. The IM guarantees forward progress of the application and also is used as a safety-net in case instructions cannot be included in basic block translations and superblocks.

There is one caveat concerning the interpretation method employed in DARCO. Due to the complex and time consuming nature of building an interpreter, we decided to use the translator provided by QEMU but instead of translating one basic block at a time, it was modified to translate one instruction at a time. Since QEMU's translator was designed with portability in mind (it supports translation from various guest to host ISAs), using it to translate just one instruction introduces high overhead. In order to accommodate the high cost of such interpretation method, an interpretation cache is used to store the interpretations. Subsequent executions of the same x86 instruction are extracted from the interpretation cache. This modification reduced the cost of interpreting an x86 instruction to some thousands of PPC instructions instead of tens of thousands. Also note that no chaining is done between interpretations.

During IM, profiling information is being collected for the targets of branches which is based on a repetition counter. When the repetition counter reaches the *BB_translation_threshold*, the ESL switches to BBM in order to translate the corresponding BB.

Note that since we use a modified version of the QEMU translator and code generator, we also inherit some of the nomenclature. The intermediate representation of the instructions in DARCO is called *qOps*.

Figure 4 shows an abstract version of a typical translation of an x86 BB. The original code is being translated into an equivalent set of qOps. ESL translates all x86 memory operations in a special way. We introduced new qOps and PPC instructions for all load and store instructions in order to be able to distinguish during the execution whether a memory access corresponds to the application itself or the ESL. There are two reasons for doing this. The first regards to functionality. The PPC component needs to know if there is an access to the x86 memory space and in the uncommon case that the data page was not communicated before, request the page from the Controller as explained before. The second reason regards to evaluation, since we would like to be aware of the performance characteristics of each translation.

At the end of the translation, two exit stubs are attached and the BB branch target is modified to point to the taken exit stub. Each exit stub consists of an empty position where the chaining will be patched later during the execution, an update of the `%eip` and branch to the ESL where the BB starting at the new `%eip` will be interpreted or translated. When the chain position is patched, the execution will not return to the ESL, but instead the next BB will be executed directly from the code cache. Finally, a new PPC instruction, *eob_x86*, is introduced. The purpose of this instruction is strictly for synchronization. In terms of timing, this instruction has no effect.

The qOps are forwarded to the code generator. There, they undergo some basic optimizations like dead code elimination and constant propagation which contribute towards reducing the number of generated instructions. Finally, the qOps are translated to PPC instructions and stored in the code cache from where they are dispatched for execution.

2.3.2 Superblocks and optimizations

During Basic-Block translation Mode (BBM), profiling information is gathered for all BBs. This information consists of repetition and edge counters. When the repetition counter reaches a predefined threshold, an event denoting the SB creation is raised. Execution is then transferred to ESL with SBM.

In Superblock/optimization Mode (SBM), the ESL generates a new SB starting from the denoted BB. The SB generation algorithm uses the control flow information gathered throughout the execution in BBM for decision making. Specifically, there are several ending conditions for a SB:

1. Probability to exit the SB before the end has to be less than a threshold
2. The outcome of the branch is not biased according to the bias threshold
3. The biased direction is the beginning of a new SB
4. The last BB in the SB has a backward branch

The translator prepares the SB in qOps and then forwards it to the optimizer.

The optimizer applies several transformations on the SB. First, the qOps are transformed into a pseudo Static Single Assignment format. This transformation significantly reduces the complexity of subsequent optimizations. Second, the forward pass applies a set of conventional single pass optimizations (copy propagation, constant propagation, common subexpression elimination and constant folding). Third, the backward pass applies dead code elimination.

After the basic optimizations, the Data Dependence Graph (DDG) is prepared. The DDG contains all the real dependencies between the instructions along with instruction latency information. The DDG is then fed to the instruction scheduler that uses a conventional list scheduling algorithm. DARCO implements an in-order processor where instruction scheduling can improve the performance significantly. Finally, the determined schedule is used by the register allocator that implements linear scan allocation algorithm.

Finally, the qOps are translated to PPC instructions and the code is stored in the code cache. The previous entry in the code cache that corresponds to the first BB of current SB is invalidated and freed for use by subsequent translations.

2.3.3 On the work

As mentioned in the introduction, the ESL was an in house development. For some other parts we reused QEMU components after heavy modification. The gluing of these components introduces unnecessary overhead that is noticeable during the execution. For example, for short applications or applications where the interpreter is used often, the code cache look up introduces high overhead since it is done for every instruction.

Furthermore, indirect branches and return instructions are known to be one of the sources of high overhead in dynamic binary translators, since they imply re-entering the runtime system and performing a look-up for the target address. Techniques like Indirect Branch Target Cache and Sieve, proposed by [5], can be employed to reduce the number of times execution is returned to the runtime.

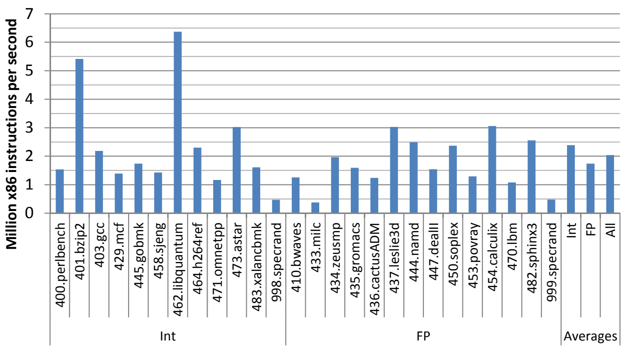

Figure 5: DARCO execution speed

Currently we are working on including control and data speculation which are techniques commonly used by co-designed virtual machines to generate better optimized code.

3. EXPERIMENTAL RESULTS

In this section we present a high level evaluation of the infrastructure and a characterization of the ESL using SPEC CPU 2006 benchmarks. Specifically, we are presenting information about the speed of the infra structure and some performance characteristics regarding ESL.

Here the results are reported in terms of *number of PPC instructions*, except figure 5 which is in terms of *x86 instructions*. Timing results are beyond the scope of this paper. The results regard the execution of the first 200 billion instructions due to simulation time constraints.

3.1 DARCO speed

The speed of the infrastructure is shown in Figure 5. We measure the speed in millions of emulated x86 instructions per second. This shows the rate of how many x86 instructions pass throughout the execution flow of DARCO which includes all the components of the infrastructure. On average the execution rate of DARCO is approximately 2 million x86 instructions per second.

The execution rate is directly affected by several parameters. For example, the performance of DARCO for 462.libquantum is the highest because it has a small static instruction footprint and small data footprint which incurs minimal communication between the various components. As another example, consider 998.specrand. The execution rate is low since the original x86 application is very small. As an effect, DARCO does not have enough time to amortize the overhead of communication between the components. Finally, consider 433.milc. The execution speed is low since a lot of data pages are communicated between the components, enforcing several synchronization points. Unfortunately, due to space limitations we cannot go into detail for all the benchmarks.

A parameter that has a definitive impact on the execution speed is the host processor. The results reported are on a

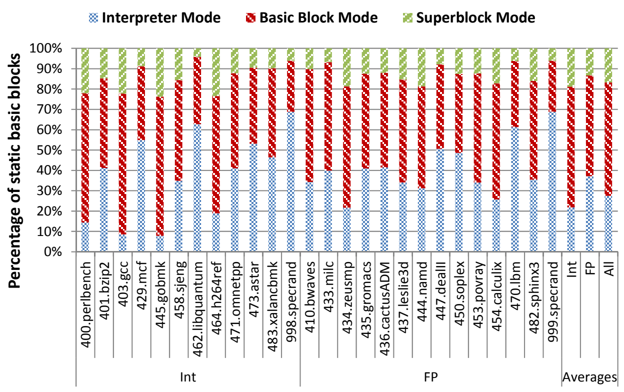

Figure 6: Distribution of static x86 basic blocks in the three modes of ESL

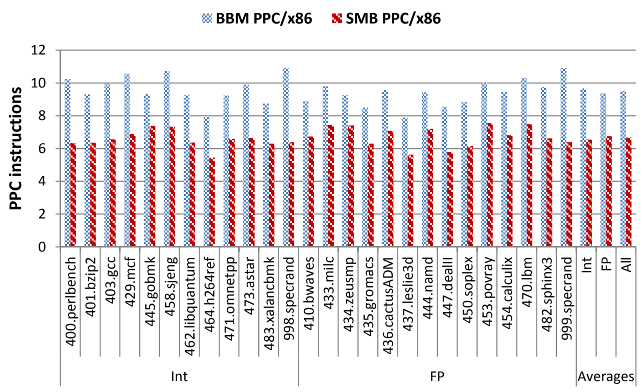

Figure 7: PPC instructions generated per x86 instructions in BBM and SBM

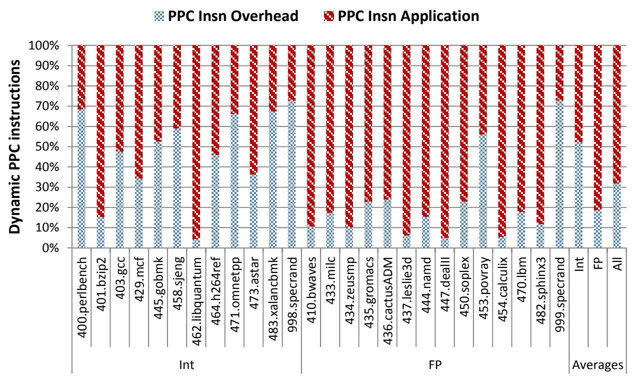

Figure 8: ESL dynamic instruction distribution

cluster where only one core is devoted per execution task. All the three processes of DARCO (x86/PPC component and controller) have to share this one only core which degrades the performance significantly. On typical dual core machines, the execution rate averages on 8 million instructions per second.

3.2 ESL and SPEC characterization

Figure 6 shows the distribution of the static x86 BB in the various modes of ESL. The threshold for promoting a BB from interpretation to BBM is 5 executions and for promoting the same BB to SBM is another 50 executions. The conclusion we can extract from this graph is that on average 18-20% of the static code is promoted to the highest optimization level which is in par with the common knowledge that ~10% of the static code of the application is responsible for 90% of the dynamic execution.

The number of PPC instructions generated for 1 x86 instruction is depicted in Figure 7. There are several things that can be observed from this Figure. First, let's concentrate on the cost of an x86 instruction in BBM. Around 10 PPC instructions are required to emulate a single x86 instruction. The cost is rather high since in the current version of ESL, the x86 EFLAGS are handled purely through

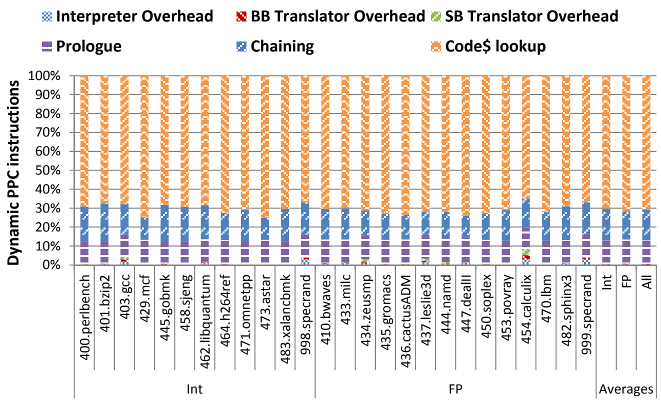

Figure 9: ESL overhead breakdown

software which introduces a relatively large number of PPC instruction for their emulation.

The cost of emulation is reduced by 30% in SBM with respect to BBM. The basic optimizations that are applied in the forward and backward pass of the optimizer are effective in reducing the amount of qOps and PPC instructions required for the emulation of x86 instructions.

In the future, we plan to improve the intermediate representation of instructions. The qOps are very generic since QEMU uses the same intermediate representation to generate code for different host ISAs. In the case of DARCO, the host architecture is specific which provides the opportunity for a targeted intermediate representation. Modifying the qOps of ESL into instructions that look closer to what the PPC ISA offers will significantly reduce the cost of emulation. Furthermore, we plan to emulate the x86 EFLAGS using hardware and provide specific hardware support for handling certain opcodes in a more efficient manner (such as calls and returns).

The dynamic instruction distribution of the ESL is shown in Figure 8. The red bar represents the amount of dynamic instruction corresponding to the emulation of the x86 application. The blue bar corresponds to the overhead introduced by the various tasks of the runtime. The introduced overhead is significantly higher for the integer applications. This is due to the extensive use of indirect branches and return instructions.

As mentioned earlier, currently the ESL does not offer any special mechanics to handle indirection. Whenever an indirect branch is encountered, execution exits the code cache and a look-up is performed to find the target BB. This is also reflected in the overhead breakdown depicted by Figure 9. The code cache lookup is the main source of overhead as it happens every time the runtime is entered.

The chaining overhead is misleadingly high (20%). The actual cost of chaining is negligible, since only a small piece of code is executed to patch the exit stubs. Most of this overhead is introduced by a non-taken branch which is there to prevent chaining during interpretation. We are considering alternatives in order to eliminate this overhead.

Furthermore, the prologue corresponds to the instructions executed to perform a "context switch" between the ESL and the application. "Context switch" refers to the necessary actions for the transition of the execution from the runtime to the application. A "context switch" takes place every time we interpret one instruction, or whenever we enter the code cache from BBM or SBM. This should also be reduced when special handling of indirect branches is introduced.

Finally, the actual overhead of interpretation, translation and optimization seems negligible when compared to the aforementioned sources of overhead.

4. RELATED WORK

Some of the most characteristic examples of process level VMs are Dynamo [15], DynamoRIO [3], IA-32 execution layer [9] and Strata [7]. All of them employ different techniques to reduce the overall overheads and guarantee that the application will reach the steady state as fast as possible. For example DynamoRIO and IA-32 EL start with basic block translation, while Dynamo starts with interpretation. Different heuristics are used to construct larger regions as early as they can afford. The common ground among the three though, is that they apply only simple, low-cost optimizations in order to minimize the overhead impacts.

In the field of co-designed VMs, where the DBO is part of the hardware platform, the most representative example is Transmeta's Crusoe [1,4] where the Code Morphing Software [4] is translating x86 instructions to a VLIW instruction set. Other examples are the DAISY/BOA [6,11] projects from IBM.

In the field of hardware-only dynamic optimizers the most characteristic examples are RePlay [12,2] and PARROT [16]. Both proposed an off-the-critical path hardware only dynamic optimizer for x86 architectures. The goal was to optimize the μops generated during the execution of the x86 application and reuse the optimized version for subsequent executions of the same piece of code.

5. CONCLUSIONS

This paper presented DARCO, a complete infrastructure that enables research on HW/SW co-designed processors. DARCO is not an early version of an envisioned infrastructure but a mature ready-to-use and debugged tool.

DARCO interprets, translates and dynamically optimizes x86 binaries in PPC instructions which execute on top of a functional PPC emulator. Its Emulation Software Layer includes an interpreter, a translator, a scheduler, a register allocator and a staged optimizer. The other key components are the controller that the user interacts with and the timing simulators.

In addition to the infrastructure, in this paper we characterize the SPEC CPU 2006 benchmarks with respect to their dynamic binary optimization behavior. We show the related overheads and analyze the optimization stages each piece of code reaches to.

ACKNOWLEDGMENTS

This work is partially supported by the Generalitat de Catalunya under grant 2009SGR1250, the Spanish Ministry of Education and Science under contracts TIN2007-61763 and TIN2010-18368, and Intel Corporation. Demos Pavlou and Aleksandar Brankovic were partially funded by the Generalitat de Catalunya with an FI-AGAUR grant. Rakesh Kumar is funded by a UPC-Research grant.

REFERENCES

[1] A. Klaiber. "The technology behind the crusoe processors". White paper, January 2000.

[2] B. Slechta et al. "Dynamic optimization of micro-operations". In *High-Performance Computer Architecture*, 2003. HPCA-9 2003. pages 165–176, Feb. 2003

[3] D. Bruening, T. Garnett, S. Amarasinghe. "An infrastructure for adaptive dynamic optimization". In *Proceedings of the international symposium on Code generation and optimization* (CGO'03), pages 265–275, 2003.

[4] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, J. Mattson. "The transmeta code morphing™ software: using speculation, recovery, and adaptive retranslation to address real-life challenges". In *Proceedings of the international symposium on Code generation and optimization* (CGO '03), pages 15–24, 2003.

[5] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, B. R. Childers. "Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems". In *Proceedings of the International Symposium on Code Generation and Optimization* (CGO '07).

[6] K. Ebcioglu, E. R. Altman. “Daisy: dynamic compilation for 100% architectural compatibility”. SIGARCH Comput. Archit. News, 25(2):26–37, 1997.

[7] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. “Retargetable and reconfigura-ble software dynamic translation”. In *Proc. of the int. symp. on Code generation and optimization* (CGO), 2003.

[8] Krewell, K., "Transmeta Gets More Efficeon", Microprocessor Report, Vol. 17, No. 10, October 2003.

[9] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, Y. Zemach. “IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems”. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture* (MICRO 36), page 191, 2003.

[10] Quick EMUlation tool. http://www.qemu.org/

[11] S. Sathaye et al. “BOA: Targeting multi-gigahertz with binary translation”. In In Proc. of the 1999 Workshop on Binary Translation, IEEE Computer Society Technical Committee on Computer Architecture Newsletter, pages 2–11, 1999.

[12] S.J. Patel and S.S. Lumetta. “rePLay: A hardware framework for dynamic optimization”. *IEEE Transactions on Computers*, 50(6):590–608, Jun 2001.

[13] Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmarks. http://www.spec.org/cpu2006/.

[14] T. Lindholm, F. Yellin. “Java Virtual Machine Specification”. Addison Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[15] V. Bala, E. Duesterwald, S. Banerjia. “Dynamo: a transparent dynamic optimization system”. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (PLDI ’00), pages 1–12, 2000.

[16] Y. Almog, R. Rosner, N. Schwartz, A. Schmorak. “Specialized Dynamic Optimizations for High-Performance Energy-Efficient Microarchitecture”. In *Proceedings of the international Symposium on Code Generation and Optimization* (CGO’04), March 2004.