DDGacc: Boosting Dynamic DDG-based Binary Optimizations through Specialized Hardware Support

Demos Pavlou[†,1] Enric Gibert Fernando Latorre Antonio Gonzalez[†]

Intel Barcelona Research Center (IBRC) - Intel Labs
[†]Department of Computer Architecture, Universitat Politècnica de Catalunya
{demos.pavlou, enric.gibert.codina, fernando.latorre, antonio.gonzalez}@intel.com

Abstract

Dynamic Binary Translators (DBT) and Dynamic Binary Optimization (DBO) by software are used widely for several reasons including performance, design simplification and virtualization. However, the software layer in such systems introduces non-negligible overheads which affect performance and user experience. Hence, reducing DBT/DBO overheads is of paramount importance. In addition, reduced overheads have interesting collateral effects in the rest of the software layer, such as allowing optimizations to be applied earlier. A cost-effective solution to this problem is to provide hardware support to speed up the primitives of the software layer, paying special attention to automate DBT/DBO mechanisms and leave the heuristics to the software, which is more flexible.

In this work, we have characterized the overheads of a DBO system using DynamoRIO implementing several basic optimizations. We have seen that the computation of the Data Dependence Graph (DDG) accounts for 5%-10% of the execution time. For this reason, we propose to add hardware support for this task in the form of a new functional unit, called DDGacc, which is integrated in a conventional pipeline processor and is operated through new ISA instructions. Our evaluation shows that DDGacc reduces the cost of computing the DDG by 32x, which reduces overall execution time by 5%-10% on average and up to 18% for applications where the DBO optimizes large code footprints.

Categories and Subject Descriptors B.8.2 [*Hardware*]: Performance and Reliability—Performance Analysis and Design Aids

General Terms Algorithms, Design, Performance

Keywords Co-designed processors; hardware acceleration; dynamic binary optimization; start-up overhead.

1. Introduction

Dynamic Binary Translation (DBT) and Dynamic Binary Optimization (DBO) by software are related techniques that have been applied (and will still be applied) for a long time for portability, backward compatibility, performance, design simplification, and virtualization, among others. DBT and DBO can be applied at different levels [22], ranging from user-level virtual machines (DynamoRIO [5], Dynamo [3], Intel's IA-32 Execution Layer [4], Java Virtual Machines [12]) to whole system virtual machines (co-designed virtual machines like Transmeta Crusoe [11], Daisy [7], BOA [18], etc.). In all these types of virtual machines a software layer (henceforth, the optimizer) is responsible to dynamically analyse code generated for a specific source ISA and translate it to a target ISA by applying several transformations and/or optimizations (the source and target ISAs may be the same). The resulting translated/optimized version of the code is often cached for reuse in a code cache.

The software layer incurs overheads since the processor spends a non-negligible amount of cycles in the dynamic process of transforming a piece of code into another, which does not allow the emulated program or system to make progress. Such overheads impact directly the overall performance of the system. However, overall performance is not the only important metric. For example, although overall overheads may be small once an application has been executed, start-up or reactive overheads may not. Start-up overheads refer to the overheads that are paid until the system reaches the steady state, where the vast majority of the executed code comes from the translated/optimized code cache. On the other hand, reactive overheads are caused by re-translation and re-optimization of regions of code that have been evicted from the translation cache, especially in the case of a multiprogrammed environment with shared translation cache. Start-up and reactive overheads are related and have an impact on another important metric: responsiveness, which in turn, affects user experience [10]. Thus, reducing the DBT/DBO overheads is of paramount importance. Furthermore, reducing the overheads has interesting collateral effects in the design of the optimizer, such as: (i) optimizations may be applied earlier or eagerly, (ii) more expensive optimizations that were discarded previously may now be applied, (iii) simpler code cache management algorithms may be employed, among others.

A key observation of DBT/DBO systems is that the software layer goes through a set of common primitives despite the level at which it operates. In that respect, the optimizer will among other things: (i) profile the code in order to detect hot regions, (ii) build regions, (iii) decode instructions, (iv) optimize regions, (v) code scheduling, (vi) code caching etc. Note that the aforementioned tasks do not need to be executed necessarily in this strict order.

In this work we characterized the overheads of a DBT/DBO system that performs several basic optimizations. From our study, we have observed that building the Data Dependence Graph (DDG) is among the most costly operations. The primary sources of overhead in a DBT/DBO system are instruction decoding, software profiling

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

VEE'12, March 3–4, 2012, London, England, UK.

and code cache. Hu et al. [10] attack the problem of instruction decoding, Merten et al. [15] propose hardware to do profiling and Hazelwood et al. [8] propose effective code cache management algorithms. In this paper, we propose to use specific hardware support to reduce the costs of building the DDG. We believe that providing specialized hardware to speedup the optimizer is the most effective solution. This is especially true in system virtual machines, in which the software layer is always executed regardless of the host applications being emulated/optimized.

In this paper, we define and evaluate DDGacc, a new functional unit, which enhances the process of computing the DDG. Computing the DDG is mainly necessary for performing a good instruction scheduling algorithm. Although we envision DDGacc to be more useful in system virtual machines, it is generic enough to be applied to a user virtual machine as well (it is mainly a matter of making the interface visible to the user). In addition, we have paid special attention to devote specialized hardware to automatize the DBT/DBO mechanisms, and leave the DBT/DBO policies to the optimizer. In this respect, the proposed hardware computes the DDG, while the heuristics to perform good instruction scheduling are left to the software layer. This is the key in order to be able to evolve/improve the optimizer over time and/or provide different optimizer configurations to different customers.

In particular, we have extended DynamoRIO with a client that applies a set of optimizations whenever a hot region is detected and a trace is created. This set consists of an algorithm for the creation of the DDG, dead code removal (DCR) and redundant load elimination (RLE). We measured the overheads introduced by such optimizations and we observed that approximately 5%-10% of the total time was spent on optimizing hot regions. The most costly optimization was the creation of the DDG which accounted for 32% of the optimization overhead. To tackle this, we propose and evaluate a hardware accelerator that provides support for the software DBO for building the DDG. Our studies show that the cost of the algorithm is reduced by 32x, using an in order processor. This is translated into an average improvement between 5% and 10% on average depending on the optimization threshold.

The rest of the paper is organized as follows. Section 2 presents the experimental framework used throughout the paper. In section 3 the overheads of the baseline DBO are exposed and characterized. The hardware accelerator we propose is presented in section 4. DDGacc is evaluated in section 5. In section 6, we discuss related work and concluding remarks are given in section 7.

2. Experimental framework

In this section we describe the DBO infrastructure we used for our studies as well as our evaluation methodology.

2.1 DBO Infrastructure

As mentioned before, we envision DDGacc to be used mainly by co-designed VMs. However, to the best of our knowledge, there is not a single public and extensible tool to perform research on co-designed VMs. Since the proposed solution can also be used by user-level VMs, we have used DynamoRIO [5], a state-of-the-art x86 user-level binary analyzer to conduct our experiments. DynamoRIO is a dynamic instrumentation tool that allows runtime code manipulation on any part of the program. It provides an interface for building dynamic tools that can be used for optimization among others. It is in fact a process VM which takes control of the application and translates basic block by basic block at the beginning. When a basic block is determined to be hot, meaning it reached a predefined threshold, a trace is generated by packing together several BBs based on the profiling information gathered during the so far execution. Traces are single-entry multiple-exit code sequences. Translated basic blocks and traces are cached in a code cache for reuse.

Table 1. Benchmarks

Benchmark	Input Size
SPEC CPU2006	Ref
Gnome Calculator	Application Start
OpenOffice Writer	Application Start
OpenOffice Calc	Application Start

For our study, we implemented a tool that applies optimizations whenever a new trace is created. Specifically, we first create the data dependence graph (DDG) and do some preparation for memory aliasing by analyzing load and store instructions. Then, we apply dead code removal (DCR) and redundant load elimination (RLE). DCR and RLE are quite traditional algorithms in the context of DBO since their overhead is small and can give significant performance benefits when applied to large regions such as the traces. On the other hand, the creation of the DDG introduces high overhead because of its complexity and large data structures that are traversed during the creation. Critical optimizations though, like instruction scheduling and hoisting, depend on the DDG creation for their effectiveness. The implemented algorithm is based on [16].

2.2 Evaluation methodology

For the evaluation of the overheads of the DBO we used the functional model of Simics to collect execution traces. The emulated machine uses an x86 processor based on Pentium4 and runs Fedora Core 5. The linux process tracker provided by Simics was used to isolate the dynamic instruction stream of the selected application (the application running under the control of DynamoRIO) and we used hooks in order to mark the code sections of interest and be able to distinguish them in the dynamic stream.

For this evaluation we picked two different types of application suites that cover the two scenarios a dynamic binary optimizer typically faces. On the one hand, we selected SPEC CPU2006 [1] suite as an example of applications with small number of static instructions with respect to the number of executed dynamic instructions (30.3M dynamic/static). Dynamic binary optimizers are very effective on these applications because any overhead produced on the optimization of a static instruction is easily amortized since this instruction is executed very often. On the other hand, we picked GnomeCalculator, OpenOffice Writer and Calc as an example of applications with large number of static instructions with respect to the dynamic number of executed instructions. This latter scenario is complex for dynamic binary optimizers because overheads spent on optimizing static instructions are sometimes not amortized incurring in poor performance. The set of applications is summarized by Table 1.

For all the benchmarks, the first 300 million instructions are being ignored in order to avoid completely the initialization phase of DynamoRIO, as well as some of the applications initialization routines. Then, we split the execution of each application into chunks of 1 billion instructions and we generate a trace for the odd chunks so that we can analyze the behavior of DynamoRIO for different phases of the applications. In total we collect the traces for 5 1-billion instruction chunks for each benchmark covering in total the first 9 billion instruction of the application.

For the timing evaluation, we implemented an x86 trace simulator which assumes a simple in-order processor. The configuration used is summarized in Table 2. We assume a perfect instruction cache due to constraints in the simulation infrastructure. The cache structures and the branch predictor used are based on the ones provided with SimpleScalar [2].

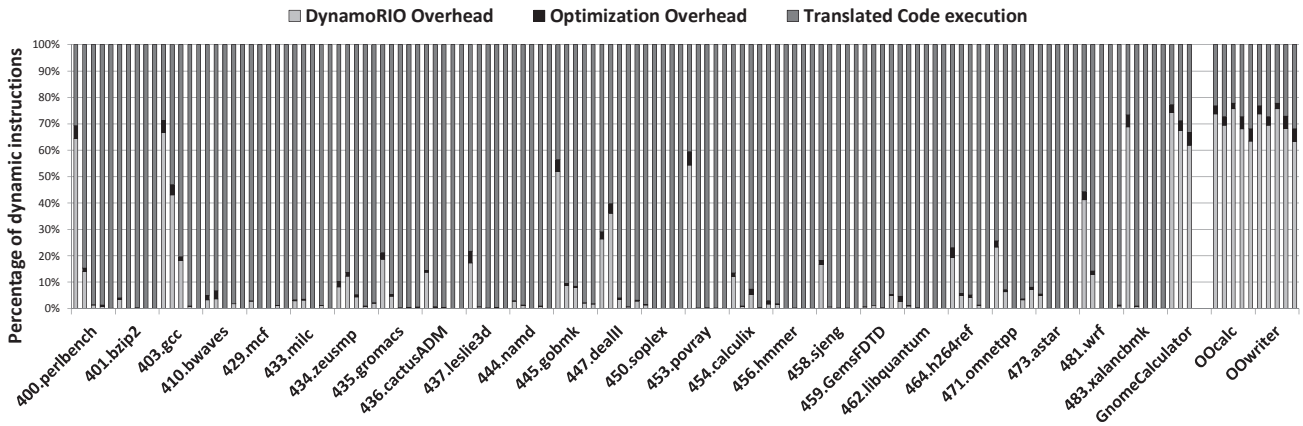

Figure 1. Dynamic instruction breakdown including DynamoRIO overhead. Each application has 5 assosiated bars, one for each 1 billion instruction chunk.

Table 2. Timing simulator configuration

Issue Width	2
Issue Queue Entries	16
Address Calculation Units	1
Simple Arithmetic Units	2
Complex Arithmetic Units	2
Branch Predictor	2K entries correlating
BTB	256 entries, 4-way
L1 Data Cache	64KB, 64B block size 4-way
L1 Data Cache latency	1 cycle
L2 Cache	512KB, 128B block size 8-way
L2 Cache latency	10 cycles
DTLB	256 entries 8-way
DTLB miss penalty	100 cycles
Memory latency	100 cycles

3. Overhead characterization

In this section we discuss the dynamic instruction breakdown of the benchmarks under DynamoRIO with the optimizing tool.

A complete breakdown is shown in Figure 1. Each bar represents one of the selected 1 billion dynamic instruction chunks. The instructions are grouped into three main categories. The first one is "DynamoRIO overhead" and it includes the process of creating basic blocks and traces, as well as the management of the code caches and the client tool. This overhead is quite big due to the fact that DynamoRIO is implemented as a generic tool that provides a rich interface to enable the user to build his/her own client. Such overheads are dominated by the insertion of callouts/trampolines, and a rather complex instruction decoding and representation in a generic and flexible intermediate language. The second one is "Optimization overhead" which includes all the instructions that correspond to the optimization process. Notice that a more detailed breakdown is presented later for this category since it is our main focus. Finally, the category "Translated code execution" represents the instructions executed from the code cache.

It is interesting to notice that most of the SPEC CPU2006 benchmarks share the same characteristics as expected. The biggest overhead is encountered at the beginning of the application, specifically in the first chunk and sometimes the second. This is justified by the high repetition of a kernel loop. Over time, the overhead introduced by the DBO is amortized.

In contradiction, for large applications like GnomeCalculator, OOcalc and OOwriter, the DBO incurs high overhead throughout the whole execution. This kind of applications have abundant static code which exceeds the optimizations threshold, while the reuse of the code is not enough to completely hide the overhead. Furthermore, as commented before, overall performance is not the only important metric, as responsiveness is often as important, which signifies the need for more efficient translation/optimization.

In the context of this paper, we are more interested in the overhead introduced by the optimizer alone. As mentioned before, the bar labeled "DynamoRIO overhead" is quite big due to the flexible nature of its design. This is so because the version of DynamoRIO we are using (the one that is publicly available with its full source code) is more an instrumentation tool than a DBO itself. Such overheads would be totally removed if we strip out such instrumentation features, as we have observed in other DBO tools such as Strata [21]. For example, since the source and guest ISAs are the same (x86), during the basic block translation phase, only partial decoding is necessary in order to distinguish branches and collect their targets. In contrast, in the presence of an instrumentation client DynamoRIO is doing a full decoding in order to facilitate code manipulation by the client. Thus, from this point forward we do not account for the DynamoRIO component in our study, as this overhead does not exist on pure DBO infrastructures. Ignoring this part of the overhead does not restrict our study since we are mainly concentrating on the overhead of the optimizing part of the VM with respect to the application's code execution.

3.1 The real view of the optimizer overheads

Consider Figure 2 which presents the breakdown of the optimizer and the translated code. The top bar shows the instructions that correspond to translated code. The second bar shows the instructions of the two basic optimizations; RLE and DCR. The third bar shows the instructions for the memory aliasing preparation and the fourth bar represents the instructions corresponding to the DDG. Finally, some other sources of overhead like the initialization phase of the optimizations are shown with the bottom bar.

A general observation is that the optimization overheads for most of the SPEC CPU2006 applications are negligible, especially after the first billion of instructions. This can be justified by the fact that 10% of static instructions are responsible for 90% of the execution and static instruction reuse is huge. There are though some

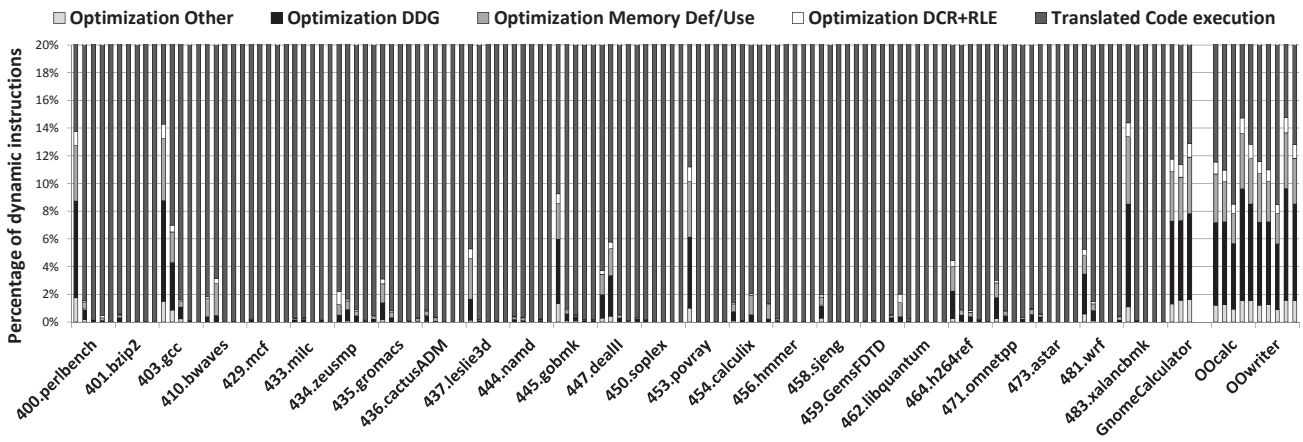

Figure 2. Dynamic instruction breakdown of optimizer and executed application. The Y axis maximum is at 20%. The rest 80% accounts for Translated Code execution. Each application has 5 assosiated bars, one for each 1 billion instruction chunk.

exceptions like 403.gcc where the execution is not concentrated in a few loops alone. This leads to the generation and optimization of more traces. Moreover, the overall overheads are low since the default threshold for creating a trace in DynamoRIO is at 50 executions, which gives a good balance between the amount of traces and coverage.

As it can be seen, applications with low static footprint are more tolerant to DBO overheads because they are amortized by the high repetition of the optimized code. However, DBO overheads for applications with large ratio between static and dynamic instructions are significant leading to poor performance and low response time.

Studying the breakdown in detail, we noticed that the DDG is the most costly part accounting on average for 6% of the dynamic instruction stream where DBO has high activity. An interesting difference is that for applications with low static to dynamic ratio, the proportional part of the DDG varies from 1% to 6% whereas on applications with high ratio the overhead varies from 5% to 7% across all the traces.

While such overhead could be avoided by a DBO that targets an out-of-order processor, computing the DDG is a key component for a DBO that targets in-order or VLIW processors, like Crusoe or Efficeon. The use of the DDG enables the implementation of more efficient and fruitful algorithms for optimizations like instruction scheduling and register allocation which are quite important for final performance. For instance, it has been shown that instruction scheduling is a key DBO component to enhance the quality of the generated code [6].

4. Reducing the overheads

Providing hardware support to speed up the execution of DBT/DBO primitives is the most efficient solution to reduce its overhead, since the runtime of the VM is always executed regardless of the emulated/translated application. However it is important to devote hardware to enhance the mechanisms of the primitives, leaving the execution of the heuristics to the software, which has more flexibility.

Moreover, one of the purposes of DBO is to provide maximum performance on simple processors like in-order processors so that we sustain good performance at a very low power [11]. For this reason, although hardware hooks are natural solutions to overcome DBO overheads, these HW enhancements should be simple enough so that we do not lose the power advantage of using a simple processor.

```
1  readOps[]
2  writeOps[]
3  for each insn i {
4      readOps[i] = extract insn read operands
5      writeOps[i] = extract insn write operands
6  }
7
8  for each insn i {
8       for each readOp in readOp[i]
9           if(readOp defined in trace)
10              add flow dependence from last definition
11      for each writeOp in writeOp[i]
12          if(writeOp defined in trace)
13              add output dependence from last definition
14          if(writeOp used in trace)
15              add anti dependence from last uses
16      Update register uses with readOp[i]
17      Update register definitions with writeOp[i]
18 }
```

Figure 3. Pseudo code of the DBO's module that creates the DDG

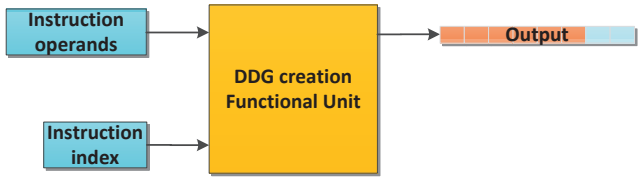

Figure 4. DDG functional unit block diagram

In this section we describe the proposed hardware support for the creation of the DDG. We take a top down approach in describing our proposal. First we discuss the software-only implementation in order to derive the basic components that are necessary for the execution of the algorithm. Such software-only implementation is our baseline. Then, we describe the proposed hardware support, its interface, and we demonstrate its functionality through an example. Finally, we explain how the functional unit can be integrated in a conventional pipelined processor.

4.1 Software-only implementation

As mentioned in section 2 the DDG creation algorithm is based on the one described in [16]. The DDG stores all the flow dependences (RAW), anti-dependences (WAR) and output dependences (WAW)

based on the register definitions and uses of the instructions belonging to a trace. All the memory operations are considered to be dependent between them, meaning that they must execute in order. We may also add hardware in our future work to perform memory disambiguation and be more aggressive with memory dependences. The pseudo code of the part of the DynamoRIO tool that implements the software-only algorithm is shown in Figure 3. The first loop traverses the instructions to extract the read and write register operands to some custom structures in order to avoid the extensive use of DynamoRIO API and increase data locality. The second loop traverses the operands that are read or written and according to the definition and use state of each register from the previous instructions, dependences are added to the graph. The actual implementation, in our case, is more complicated since there are several corner cases due to the x86 particularities that need to be taken in to account. One such case is register aliasing. For example eax, ax, al, ah are all the same register but when al or ah are defined the rest of the register remains unchanged. The def/use state consists of an array with an entry for each architectural register that holds the last definition of that register and the uses corresponding to it.

4.2 Proposed solution

As shown in section 3, the creation of the DDG is the most costly primitive compared to the rest of the optimizer. In order to improve the efficiency of the DBO we propose a functional unit, called *DDGacc*, that could be easily integrated in the processor pipeline. Such a functional unit will work in a similar manner as the software implementation, meaning that the DDG is going to be built incrementally. The block diagram of the *DDGacc* is depicted in Figure 4. The input to the FU is the operands of one instruction and the index of the instruction inside the corresponding trace. The input will be given by means of a new ISA instruction. The output will be the indexes of the instructions from which the input instruction has dependences from.

In more detail, the input of the accelerator is a 32-bit value that describes a single instruction. Its fields are shown in Figure 5(a). The first part is composed by all the register operands of the instruction, either explicit or implicit. Seven bits are used to represent a register id (Reg1, Reg2 and Reg3) and two extra to determine if the register is read, write, read/write by the instruction or not used (invalid). Note that due to particularities of the x86 ISA, an instruction can have more than one destination register (e.g. `xchg` and `pop`) which deems necessary to mark all the registers with the two extra bits. Moreover, a single bit is used to designate if the instruction is a memory read (load) and another if it is a memory write (store). Finally, two extra bits are used to designate whether the `EFLAGS` register is modified or read following the same definition as Reg Type field. Adding up, the inputs consist of 31 bits which is convenient since a general purpose 32bit register can be used to carry the input. Note that if the target architecture provides 64bit registers, more detail can be provided to the functional unit, e.g. separation of the `EFLAGS` in three groups and/or the input traces can be larger than 256 instructions.

The output of the functional unit is depicted in Figure 5(b). Each element contains the index of the instruction from which the input has flow or output dependence. Since we limit our traces to 256 instructions, 8 bits are sufficient to encode the index of any instruction in the trace. For each register operand specified in the input, one flow dependence field is pre-assigned in the output. For example if there is flow dependence due to Reg1 then the index of the instruction defining Reg1 will be written into the Reg1 output field. The fourth and fifth fields (MemW, MemR) are used for flow dependences due to memory operations. Finally, the last two elements are used to track output dependences. Note that the last element can also be used for output dependences due to the `EFLAGS` register, since instructions that define two registers at the same time like `xchg` do not modify the `EFLAGS`.

4.3 DDGacc operations

The DDGacc will be operated by means of new ISA instructions. Specifically two new instructions are required. The first one is an instruction that will initialize all the internal structures of the accelerator in order to be prepared for the creation of a new DDG. The instruction is named ***ddgReset***.

The actual dependence calculation will be instructed using the second ISA instruction called ***ddgAppend*** which has the format: `ddgAppend m64, Rin`.

The instruction is similar to a store instruction where m64 is the memory position where the output of the functional unit will be stored. The address is described by [DDGBase@+insnIndex] where DDGBase@ is the beginning address of the output array, which can be held in a general purpose register, and insnIndex is an offset computed by the functional unit which is internally incremented each time ***ddgAppend*** instruction is executed. The input is given by `Rin` which is a general purpose register with values as described previously. The instruction operates like a conventional store with the only difference that the data will be taken directly from the functional unit by means of forwarding the output value directly to the store buffer just like a normal `add mem, reg`.

As an example, consider the implementation of the DDG algorithm depicted in Figure 6(a) and compare it to that in Figure 3. The basic difference from the software-only algorithm is that the dependences and the tracking of the definitions of the registers are done by a single instruction that will instruct the functional unit to handle them. In the example given in Figure 6(b) we apply the algorithm on a sample trace of 3 instructions. Concerning the input, the first instruction defines and uses register ebx, hence the first register is marked read/write. Moreover the add instruction modifies the `EFLAGS` register. The second instruction defines `eax` and reads register `ebx`. The last instruction stores the value of `eax` to the address contained by `ebx`, therefore the write memory field is marked with 1. As for the output, the first instruction has no dependences since it is the first in the trace. The second instruction depends on the first one because it uses `ebx`. The last instruction depends on both previous since it uses the two defined registers.

4.4 DDGacc implementation

The functional unit design is depicted in Figure 7. Due to space constraints, we do not describe here the detailed implementation. We refer the reader to a technical report [19] for further details.

The proposed accelerator contains a small memory structure, called LDRF that keeps track of the last definition of each register. Each line of the structure has size of 8 bits, since the maximum index number encountered is 255, and 92 lines are needed to store the index of the defining instruction for each register. In total, the memory needed to handle the registers is 92 Bytes plus 92 valid bits. The lastMemoryW, lastMemoryR and the lastEFLAGSW keep track of the last instruction that was a memory write, a memory read and that modified the EFLAGS register respectively, increasing the total memory requirements to 116 Bytes.

The LDRF is indexed by the 3 register IDs given from the input. Three read ports are needed to get the last definition of all registers in parallel and two write ports to update the last definition of up to two registers in parallel. One valid bit for each entry designates whether this operand was defined by an instruction in the trace under consideration. The output of LDRF for each register is the 8 bit identifier of the defining instruction along with the valid bit (total 9 bits). The memory cells for the last memory write, read and EFLAGS definition also follow this concept. The outputs of the LDRF and the other memory cells are directly connected to their

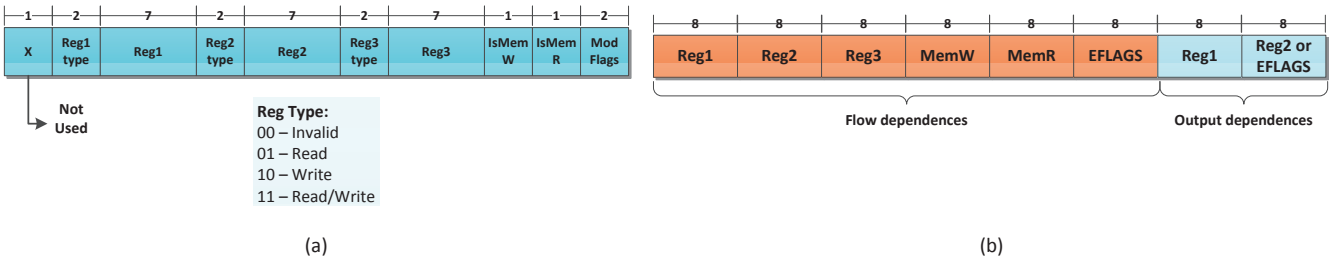

Figure 5. (a) Input of the accelerator consisting of the register operands and (b) Output of the accelerator consisting of the indexes of the instructions that there is dependence from

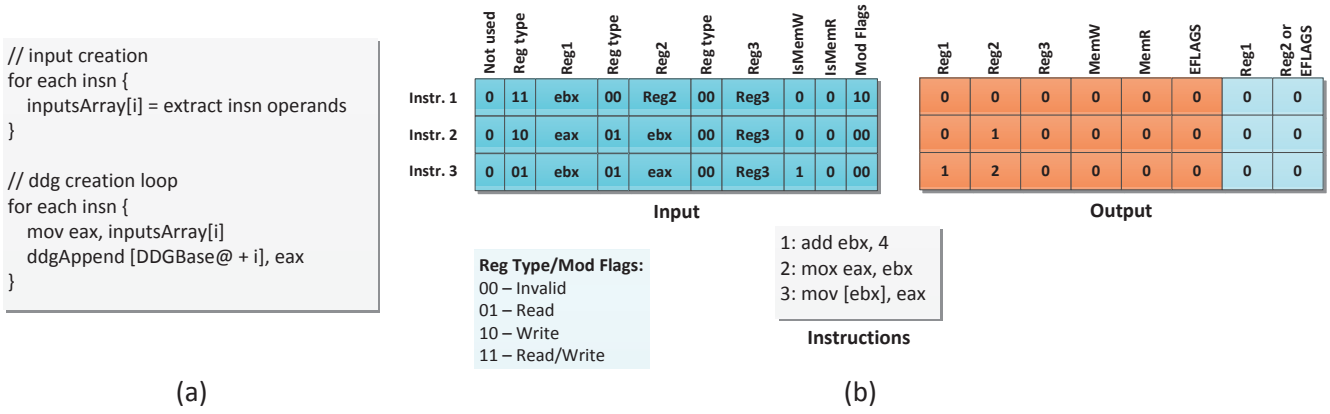

Figure 6. (a) HW/SW co-designed implementation of DDG algorithm. (b) Example of operation on instructions

respective output slots that will be written to memory. The final outcome depends on the different control bits from the input.

Internally to the functional unit, a two-step process is necessary for it to operate correctly: read and update. In the first step, the values from the LRDF and the other memory cells are read and the output is prepared. On the second step, the defined operands have to be updated with the index of the current instruction.

We have identified that the LDRF structure is the critical path of the proposed functional unit, which can be implemented as a 3-read and 2-write direct-mapped cache. We have calculated the access time and the area of LDRF using CACTI [24] assuming 32nm technology.

The access time for such small cache is 0.19ns. Since the tag and the data are accessed in parallel, we can safely assume that this access time is an upper bound for the access time of the LDRF. In reality, the LDRF access time will be less. Assuming a cycle time of 0.33ns (3 GHz) then both steps (read and update) can be done in two cycles. Finally, the area required for this structure is 0.003495 mm2.

Regarding the LDRF initialization with the ddgReset instruction, in order to minimize the penalty of initializing the LDRF, the valid bits can be gang-cleared as explained in [14]. The latency of this instruction will be one cycle.

Another issue is whether the memory kept in the functional unit should be part of the architectural state. The state of LDRF and the rest of memory cells are vital for the correctness of the output. The answer depends on how these instructions are going to be used. If an algorithm using these instructions is implemented as a part of the DBO in a co-designed virtual machine, the state is not necessary to be part of the architectural state as long as the VM guarantees that the algorithm cannot be interrupted. On the other hand, if the instructions are used in the context of a software-only DBO, complete execution is not guaranteed. Hence, in this case the state of the LRDF and the other memory cells has to be part of the architectural state.

The parameters used for the description of DDGacc are targeted towards x86 ISA, due to our evaluation infrastructure. We would like to point out that use of the functional unit is not limited by any manner just to x86 ISA. In fact, the changes that are necessary for retargeting are very few and it could be also enhanced for providing more information with the output if the ISA supports for example 128bit stores.

5. Evaluation

This section is organized in two parts. In the first part we present and discuss the evaluation results of our proposal against the baseline software-only DBO. For the timing evaluation, we selected the chunks in which more traces were generated and optimized. We present results for a total of 24 chunks of 1 billion instructions. For the second part, we show the effectiveness of the accelerator when the threshold for generating and optimizing traces is lower. For all the experiments, the latency of the accelerator is assumed to be 2 cycles as explained in the implementation section. However, it is worthy to note that results obtained with larger latencies show that the proposed solution is extremely tolerant to latency.

In this paper we concentrate on reducing the introduced overheads of the optimization sequence rather than evaluating the performance gains from applying different optimizations. As such, we do not present results comparing the performance of the application executed under the DBO against native execution.

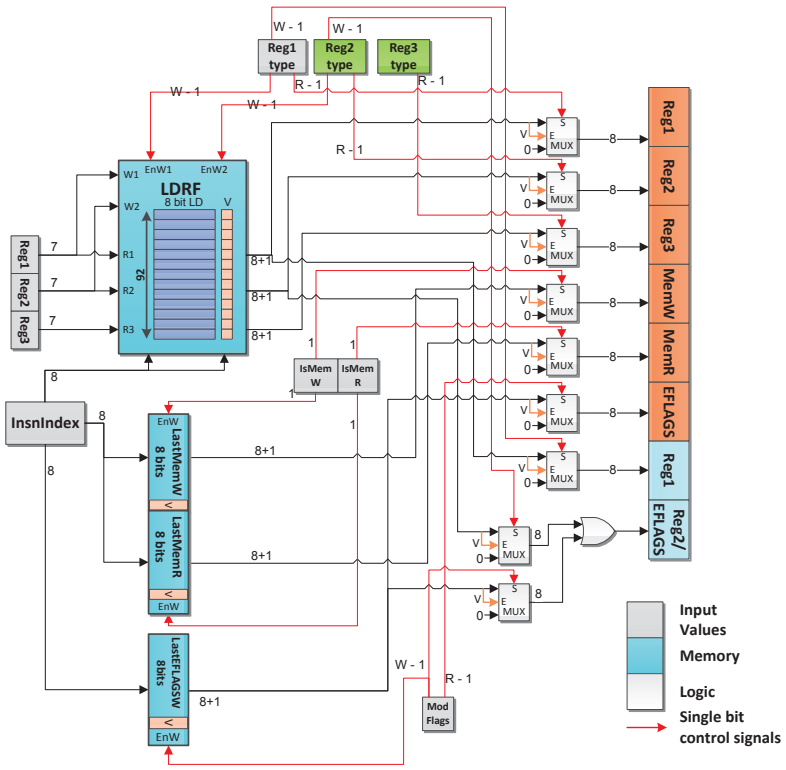

Figure 7. DDGacc implementation

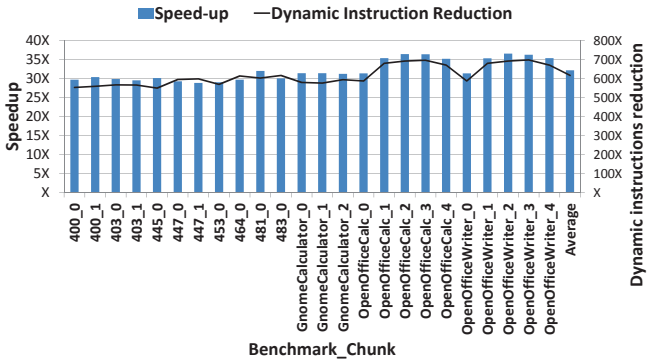

Figure 8. Speedup of the HW/SW co-designed DDG against the software-only implementation and the reduction of the dynamic instructions

5.1 Evaluation against the baseline

Figure 8 shows the speedup of the hw/sw co-designed algorithm to compute the DDG against the software-only implementation. Moreover, on the secondary axis we show the reduction of the dynamic instructions corresponding to the DDG creation. On average, the speedup is 32x while the dynamic instruction reduction is 615x. These big benefits can be understood by comparing the software only algorithm presented in Figure 3 to the proposed hardware/software algorithm shown in Figure 6(a)

However, there are several reasons why a 600x reduction in the number of dynamic instructions is not translated into a 600x reduction in number of cycles. First of all is that we assume a perfect instruction cache, which means that the reduction of instruction cache accesses due to less dynamic instructions is not reflected in our results. Cycle reduction results would be better if an instruction cache were modeled, as the software-only baseline would perform worse. As mentioned before, not being able to model an instruction cache is a caveat of our infrastructure.

Moreover, we use fixed addresses for the memory locations of the input array and the DDG array for the hw/sw algorithm. This results in an increased miss rate on the L1 D$ compared to the software-only algorithm, in which locality is exploited. This reduces the performance benefits. This is also a caveat of our research infrastructure.

Concerning the variation of the speedup and the dynamic instruction reduction among the different chunks (speedups ranging from 28x to 36x and reductions from 550x to 696x), the cost of calculating the dependences of a static instruction, with the software algorithm, is not constant. The cost varies with respect to the type of the instruction, the number of register operands (explicit and implicit) as well as the type of the register. For example the cost of an instruction that is using an aliased register (e.g. eax) is higher since all the "sub-registers" must be taken into consideration. On the other hand, the hardware accelerator is handling the aliasing through its normal functionality.

Figure 9 shows the execution time of both implementations normalized to that of the baseline. The bar labeled as SW refers to the software only implementation and the bar labeled HW/SW refers to our proposal. The total execution time is divided into three regions. The first one stands for the optimization overhead excluding the DDG generation (blue bar) which is represented by the second region (red bar). The third one is the time spent executing instructions from the translation cache (green bar). The software only implementation of the DDG algorithm accounts for 4.6% of the total execution time on average and for 32% of the optimization time. The overhead varies from 1%, for traces with

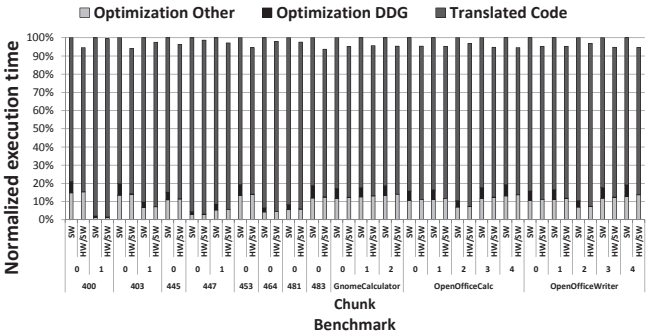

Figure 9. Normalized execution time breakdown of the baseline DBO and the DBO using the hw/sw co-designed DDG creation

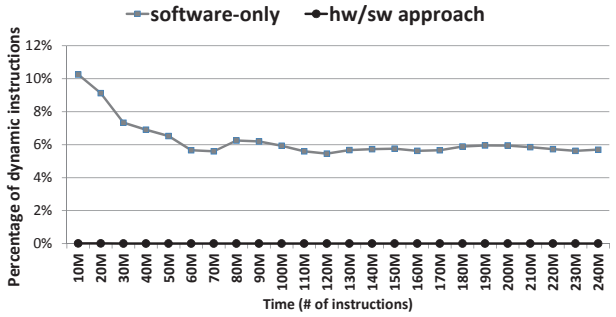

Figure 10. DDG activity over time

low DBO activity like 447.0, to 7% for traces with high DBO activity like OOWriter.4. While the dynamic instructions of the DDG are higher than the rest of the optimizations, the fact that in the software version data locality is well exploited, the execution time is lower.

Using the hardware functional unit for the creation of the DDG, improves the execution time significantly. The section that corresponds to DDG is reduced to 0.1% of the execution time across all traces which is almost negligible. The total execution time is reduced by an average of 4.6% due to this reduction, while on the traces with high activity of the DBO the benefits are up to 7%.

The direct benefit from this improvement is that the overhead introduced from the optimizer is reduced by approximately 27%. The execution time gained could be used to apply some other complex optimization which will be using the DDG or as a raw decrease of execution time. We believe that this improvement is of high importance in the context of co-designed VMs, since the impact of the overhead is significant not only at the beginning of the execution of a process, but throughout its life since evictions from the code cache can result in re-translation and re-optimization of regions of code.

In addition, there are side effects due to this reduction. The 615x reduction of dynamic instructions results to fewer accesses to the instruction and data caches. We observed 417x reduction of data cache accesses. Moreover, this reduction of instructions results in less energy consumption since fewer instructions need to be executed.

Finally, Figure 10 shows the optimizer activity over time with respect to the DDG creation. In particular the y-axis shows the accumulated ratio of dynamic instructions executed to compute the DDG with respect to the rest of the instructions over time

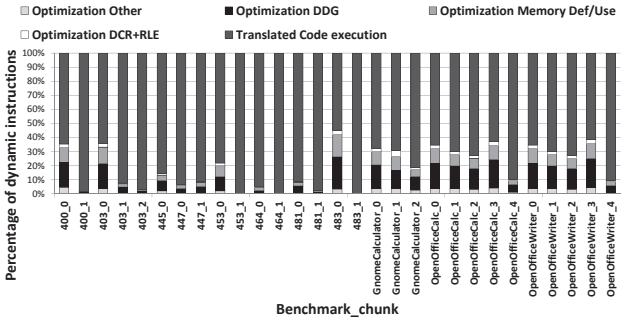

Figure 11. Dynamic instruction breakdown of the selected chunks when optimizing at the threshold of 10

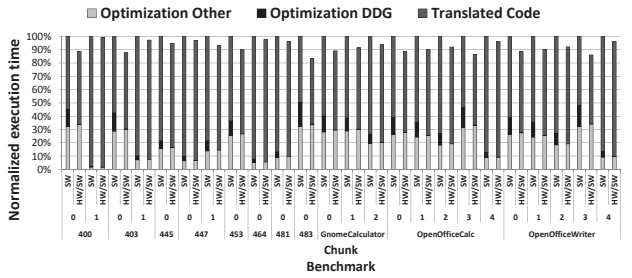

Figure 12. Normalized execution time of the baseline DBO with threshold of 10 and the DBO using the hw/sw co-designed DDG creation

(the x-axis is the 1 billion instruction chunk divided into sub-chunks of 10M instructions each). We present the results for the first chunk of the application GnomeCalculator as an example. As the applications progresses with its execution, less code is being optimized, which means that the overheads devoted to compute the DDG are lower. The line of the software-only approach shows this clearly with an initial overhead of 10% which is reduced to 6% towards the end of the chunk. Although the hw/sw approach has the same trend, the introduced overheads are negligible which makes our approach very intriguing.

5.2 Applying the optimizations earlier

Following our initial claim that using the functional unit will reduce the overhead of the DDG creation algorithm to a point that it can be applied earlier, we reduced the threshold at which traces are being generated from 50 repetitions to 10. Of course this means that more traces will be generated and optimized, resulting to higher overhead, but optimizing the instructions earlier would also imply higher performance and better response time.

The breakdown of the dynamic instruction stream for the 24 selected chunks is depicted in Figure 11. In general the overhead is increased because of the increase of the optimizer's activity, which for some applications is significant, e.g. for the real applications where it doubles. Regarding to SPEC applications, for some of them like 400 and 403 it is increased significantly as well. On the other hand, applications like 464 and 481 where the overheads where quite low with a threshold of 50, there is no substantial increase of the overhead since the amount of static code optimized is almost the same. In these cases, code is optimized earlier with a lower threshold which is the desired effect.

Considering execution time, the breakdown is depicted in Figure 12. The software implementation of the DDG accounts for 8%

of the execution time on average for the SPEC applications and 12% for the real applications with a maximum reaching 18%. For the global picture, overhead can be as high as 50% of the execution time. Such extreme overhead is what is preventing early optimization.

With the use of DGGacc, the DDG creation overhead is reduced to 0.2% of the total execution time, providing a global speedup of 8% for the SPEC applications and 12% for the real applications. The overhead is again reduced by approximately 27%. Such improvement is more significant in raw numbers than when optimizing at the threshold of 50 since it allows for the DDG creation to be applied as early as possible without introducing additional overheads.

The ramification of such low execution time requirements is that not only the optimization can be applied earlier at the beginning, but it can also be applied earlier when a hot region is evicted from the code cache in the case where it is shared by several threads. Earlier optimization will allow the application to reach a steady state earlier which will result in more repeated executions of the optimized code. This in turn will allow for the overhead to be amortized earlier and the performance benefits to be observed as soon as possible.

6. Related work

Some of the most characteristic examples of process level VMs are Dynamo [3], DynamoRIO [5] and IA-32 execution layer [4]. 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. Our proposal offers a viable example that hardware acceleration could in fact enable more complex optimizations in these systems.

In the field of co-designed VMs, where the DBO is part of the hardware platform, the most representative example is Transmeta's Crusoe [11] where the Code Morphing Software [6] is translating x86 instructions to a VLIW instruction set. Other examples are the DAISY/BOA [7, 18] projects from IBM. Both of them apply a set of low cost optimizations in order to increase the ILP and take advantage of their underlying VLIW architectures. There is no report of any hardware accelerators for their optimizations.

Characteristic examples of hardware only optimizers are rePLay [17] and PARROT [20]. Both of them implement an optimization pipeline which operates on the ìops generated during the execution of x86 code. rePLay focuses on performance gains while PARROT is giving special attention to power awareness. The major drawback of hardware-only implementation of optimizations is that the mechanism and the heuristics assumed are hardcoded. In contrast, our proposal is providing hardware support for the primitive of the optimization but leaves the heuristics to the software which has more flexibility.

Adore [13] and Trident [25] also use hardware support for dynamic optimization. Specifically, Adore uses hardware performance counters to collect samples for profiling and when it detects a hot region a second thread is invoked to generate and optimize the hot region. Trident works in a similar manner while utilizing multiple helper threads, hardware events and simultaneous multithreading to amortize the introduced overhead.

There are several other proposals (either software or hardware) that focus on speeding up basic actions of the DBT/DBO. Instruction decoding was attacked by [10]. Hardware based profiling was proposed by [15]. Effective code cache management algorithms were proposed by [8] and by [23]. The implementation of Strata, a retargetable and reconfigurable software dynamic same-ISA translator is described in [21] and techniques for reducing the overhead of indirect branches is proposed in [9]. Other tasks such as interpretation, trace chaining etc. are discussed in [22].

7. Conclusions

DBO overheads may turn into poor performance and slow response time for applications with large static code. Reducing the overheads of the software layer (the optimizer) in a DBT/DBO system impacts overall performance and user experience. Furthermore, reduced overheads have interesting collateral effects that may affect the design of other optimizer components, such as applying optimizations earlier or eagerly, among others. For all these reasons, we propose to provide hardware support to speed up the mechanisms of DBT/DBO primitives as a solution, leaving the heuristics to the software, which has greater flexibility.

In this paper, we have characterized the overheads of a DBO system based on DynamoRIO, implementing several basic optimizations, and we have identified the biggest sources of overheads. We have observed that the construction of the Data Dependence Graph (DDG), a crucial primitive to enable other optimizations such as instruction scheduling, accounts for 5%-10% of the execution time.

Hence, we have proposed hardware support to accelerate this part. DDGacc is integrated in a conventional pipelined processor as a regular functional unit that is operated by two new ISA instructions. We have provided a detailed description of DDGacc, its interface and how software can use it. After that, we have compared the performance of the proposed hardware/software solution against a software-only implementation, and we have observed that the cost to compute the DDG is reduced by 600x in number of dynamic instructions and 32x in number of cycles. This is translated on an overall improvement of the system of 5%-10% on average, depending on the optimization threshold. Moreover, this technique provides up to 18% for applications with large static code footprint.

Acknowledgments

The presented work was 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 was partially funded by the Generalitat de Catalunya under an FI-AGAUR grant.

References

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

[2] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59 –67, feb 2002. ISSN 0018-9162. doi: 10.1109/2.982917.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: http://doi.acm.org/10.1145/349299.349303.

[4] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. Ia-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium®-based systems. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 191, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2043-X.

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

275, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X.

[6] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X.

[7] K. Ebcioğlu and E. R. Altman. Daisy: dynamic compilation for 100% architectural compatibility. *SIGARCH Comput. Archit. News*, 25(2):26–37, 1997. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/384286.264126.

[8] K. Hazelwood and M. D. Smith. Managing bounded code caches in dynamic binary optimization systems. *ACM Trans. Archit. Code Optim.*, 3:263–294, September 2006. ISSN 1544-3566. doi: http://doi.acm.org/10.1145/1162690.1162692. URL `http://doi.acm.org/10.1145/1162690.1162692`.

[9] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 61–73, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2764-7. doi: http://dx.doi.org/10.1109/CGO.2007.10.

[10] S. Hu and J. E. Smith. Reducing startup time in co-designed virtual machines. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 277–288, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2608-X. doi: http://dx.doi.org/10.1109/ISCA.2006.33.

[11] A. Klaiber. The Technology Behind the Crusoe Processors. White paper, January 2000.

[12] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0201432943.

[13] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 180–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2043-X. URL `http://dl.acm.org/citation.cfm?id=956417.956549`.

[14] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: checkpointed early resource recycling in out-of-order microprocessors. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 3–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. ISBN 0-7695-1859-1.

[15] M. C. Merten, A. R. Trick, E. M. Nystrom, R. D. Barnes, and W.-m. W. Hmu. A hardware mechanism for dynamic extraction and relayout of program hot spots. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 59–70, New York, NY, USA, 2000. ACM. ISBN 1-58113-232-8. doi: http://doi.acm.org/10.1145/339647.339655. URL `http://doi.acm.org/10.1145/339647.339655`.

[16] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-55860-320-4.

[17] S. Patel and S. Lumetta. rePLay: A hardware framework for dynamic optimization. *Computers, IEEE Transactions on*, 50(6):590–608, Jun 2001. ISSN 0018-9340. doi: 10.1109/12.931895.

[18] S. S. Paul, P. Ledak, J. Leblanc, S. Kosonocky, M. Gschwind, J. Fritts, A. Bright, E. Altman, and C. Agricola. 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.

[19] D. Pavlou, E. Gibert, F. Latorre, and A. Gonzalez. Improving Dynamic Binary Optimizers Efficiency through Specific Hardware Support. Technical Report UPC-DAC-RR-ARCO-2009-11, Universitat Politecnica de Catalunya, Department of Computer Architecture, September 2009.

[20] R. Rosner, Y. Almog, M. Moffie, N. Schwartz, and A. Mendelson. Power awareness through selective dynamically optimized traces. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 162–173, June 2004. doi: 10.1109/ISCA.2004.1310772.

[21] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X.

[22] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. ISBN 1558609105.

[23] W. Srisa-an, M. B. Cohen, Y. Shang, and M. Soundararaj. A self-adjusting code cache manager to balance start-up time and memory usage. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 82–91, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-635-9. doi: http://doi.acm.org/10.1145/1772954.1772968. URL `http://doi.acm.org/10.1145/1772954.1772968`.

[24] S. Wilton and N. Jouppi. Cacti: an enhanced cache access and cycle time model. *Solid-State Circuits, IEEE Journal of*, 31(5):677–688, May 1996. ISSN 0018-9200. doi: 10.1109/4.509850.

[25] W. Zhang, B. Calder, and D. M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 87–98, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2429-X. doi: http://dx.doi.org/10.1109/PACT.2005.7. URL `http://dx.doi.org/10.1109/PACT.2005.7`.