Reducing Startup Time in Co-Designed Virtual Machines

Shiliang Hu
Dept. of Computer Sciences
University of Wisconsin – Madison
shiliang@cs.wisc.edu

James E. Smith
Dept. of Electrical & Computer Engineering
University of Wisconsin – Madison
jes@ece.wisc.edu

Abstract

A Co-Designed Virtual Machine allows designers to implement a processor via a combination of hardware and software. Dynamic binary translation converts code written for a conventional (legacy) ISA into optimized code for an underlying implementation-specific ISA. Because translation is done dynamically, an important consideration in such systems is the startup time for performing the initial translations.

Beginning with a previously proposed co-designed VM that implements the x86 ISA, we study runtime binary translation overhead effects. The co-designed x86 virtual machine is based on an adaptive translation system that uses a basic block translator for initial emulation and a superblock translator for hotspot optimization. We analyze and model VM startup performance via simulation. We observe that non-hotspot emulation via basic block translation is the major part of the startup overhead.

To reduce startup translation overhead, we follow the co-designed hardware / software philosophy and propose hardware assists to dramatically accelerate basic block translations. By combining hardware assists with balanced translation strategies, the co-designed translation system reduces runtime overhead significantly and demonstrates very competitive startup performance when compared with conventional processors running a set of Windows application benchmarks.

1 Introduction

Co-designed virtual machines (VMs) [10, 21, 22] provide the processor designer with new opportunities for innovation through the combined, close interaction of implementation level hardware and software. The overall design supports a conventional (legacy) ISA, but the hardware directly supports an implementation ISA that is designed for superior performance and/or power efficiency. A concealed layer of software mates the conventional ISA with the new implementation ISA. This layer of software not only translates the conventional binary to the implementation ISA, but also optimizes the binary based on runtime-collected profile information. The hardware and concealed software are developed concurrently, i.e., *co-designed*, as part of a unified design effort.

Software-implemented dynamic binary translation and optimization are base technologies for co-designed VMs. Optimum performance is achieved only after translation and optimization have been performed, i.e., in the *steady state*. On the other hand, because optimization is performed dynamically, one must also consider the time it takes to generate the translated, optimized code, i.e., the *startup time*. If a well-behaved compute-bound program is run for a long period of time, then the steady state performance will dominate and startup time is insignificant. However, there are situations where the startup time may be important, and it is this aspect of co-designed VM implementations that is of interest to us in this paper. Our goals are to (1) illustrate how runtime binary translation overhead affects co-designed VM startup performance, and (2) propose simple hardware mechanisms for reducing binary translation overhead.

To investigate dynamic binary translation, we have developed a binary translation system for a specific co-designed x86 virtual machine. It is an adaptive translator that uses a simple basic block translator for initial code emulation and a superblock translator/optimizer for emulating *hotspot* code (frequently executed code segments). We first discuss and evaluate co-designed VM performance under transient (startup) conditions via experimental simulations. These initial results indicate that basic block translation overhead is the major component of startup overhead, and hotspot optimization overhead can further exacerbate startup delays for some applications.

Then we propose two hardware mechanisms (assists) for reducing the startup time; both mechanisms are targeted primarily at basic block translation. The first of these hardware assists is a dual mode decoder at the pipeline frontend, and the second is a special-purpose functional unit added to the superscalar processor backend. Using either the frontend or the backend approaches can dramatically reduce the startup time. Through simulations, we demonstrate that with basic hardware support, a co-designed VM system can provide competitive startup performance with a conventional superscalar processor design.

1.1 Co-Designed VMs and Startup Performance

The traditional ISA used for software binary compilation and distribution is called the *architected ISA*; the most prominent nowadays is the x86 ISA. In a co-designed VM, a separate ISA, the *implementation ISA* is implemented in the hardware and can be designed in an implementation-dependent way to realize performance, power and/or efficiency advantages. The mapping from the architected ISA to the implementation ISA is performed by a concealed layer of software, the dynamic binary translator (DBT). The DBT is designed along with the implementation ISA and the hardware. We sometimes refer to the layer of concealed software as the *virtual machine monitor* (VMM). Because the implementation ISA is implemented directly on the hardware, we often refer to it as the *native* ISA, and we refer to the individual instructions as *micro-ops* because of their similarity to the micro-ops used in a conventional x86 design.

As discussed above, a co-designed VM implementation allows greater flexibility for realizing microarchitecture innovations, leading to high performance and/or energy efficient execution of conventional binary code in the steady state. However, the VM paradigm also introduces runtime software overhead that can offset performance gains achieved by the translated code.

Typically, the overhead of an optimizing DBT for each architected ISA instruction is on the order of thousands of instructions. For example, DAISY [10] is reported to take more than four thousand operations to translate and optimize one PowerPC instruction for its VLIW engine. The translation of each Alpha instruction to a proposed superscalar-like ILDP ISA takes more than one thousand Alpha instructions [19]. Because of the heavy DBT optimization overhead, VM systems usually employ a staged approach to ISA emulation. During program startup, one-instruction-at-a-time interpretation or simple basic block translation are first used. However, the emulation speed of an interpreter is typically 10X to 100X slower than native execution. An alternative (sometimes an addition) to interpretation is simple basic block translation (BBT) where code is translated one basic block at a time without optimization and is placed in a code cache [1, 3, 10] for repeated reuse. For many ISAs, the translation overhead for simple BBT is generally not much slower than interpretation, so most recent binary translating systems skip interpretation and immediately begin execution with simple BBT; the Intel IA-32 EL [3] uses this approach, for example. In our work, we also adopt this approach.

During the startup phase, profiling is used to identify hotspots, and DBT *optimization* is only applied to the hotspots. As observed in related projects [1,2,3,10], a staged emulation strategy reduces the runtime translation overhead, but it still can lead to slow start-up times when compared with conventional processors. We provide some data in Section 3 to illustrate this effect.

Clearly, long-running applications with small, stable instruction working sets can benefit from the co-designed VM scheme even when there is a significant startup overhead. However, there are important cases where slow startup can put a co-designed VM at a disadvantage when compared with a conventional processor. For example,

- *Workloads consisting of many short-running programs or fine-grained cooperating tasks*: execution may finish before the startup overhead can be fully amortized.
- *Real-time applications*: real-time constraints can be compromised if any real-time code is not translated in advance and then has to go through the slow startup process.
- *Multitasking server-like systems:* for large working-set workloads, the slow startup process can be further exacerbated by frequent context switches among resource-competing tasks. A limited code cache size can cause hotspot re-translations when a switched-out task resumes.
- *OS boot-up or shut-down*: here performance is important for many client side platforms such as laptops and mobile devices.

1.2 Related Work

There are a variety of systems that employ software interpretation and/or dynamic binary translation as their key enabling technology [1]. The Transmeta Crusoe processor [21] is a co-designed VM that uses an interpreter to emulate x86 code initially. Later versions of the IBM DAISY [10] also interpret PowerPC instructions before invoking "tree-region" translation. The Transmeta Efficeon processors [22, 27] use a staged translation strategy (4-stages including the initial interpretation) to provide the most performance-efficient optimization on each piece of code. In our research, we use a two-stage strategy, employing basic block translation (BBT) and superblock [17] translation/optimization (SBT). (Terminology: DBT is the generic term that includes both BBT and SBT as special cases).

With regard to special support for binary translation, Efficeon designers implemented an *execute* instruction that allows native VLIW instructions to be constructed and executed on the fly. This capability was added to improve the performance of the CMS interpreter [22]. However, details about the design are not published. In contrast, we propose special hardware to accelerate BBT and then save the translated code in a code cache for reuse. The rePLay [24] and PARROT [25] projects employ hardware hotspot detector to find program hotspots. Once a hotspot is detected, it is optimized via hardware and stored in a small on-chip frame/trace cache for optimized hotspot execution. The Instruction Path Coprocessor [6] is a programmable coprocessor that optimizes core processor's instructions to improve execution efficiency. In this work, we explore hardware assists (integrated into pipeline) that require simpler hardware than a full-blown coprocessor or hardware optimizer; also, in our work translated code is held in a main memory code cache.

On-the-fly profiling is an important part of DBT systems, both for identifying hotspot code and for assisting with certain optimizations. To reduce runtime overhead of profiling, there are proposals for hardware support of profiling and/or hotspot detection, An example is ProfileMe [7]. Merten *et al.* [23] proposed a 4K-entry branch behavior buffer (BBB) located after the instruction-retire-stage to identify dynamic hotspots. In our VM systems, we rely on such a hardware mechanism to detect hotspots and reduce initial emulation overhead.

IA-32 EL [3] is a software approach for supporting x86 applications on Intel IPF platforms. It dynamically translates x86 instructions into IPF VLIW instructions for user mode applications. It is a two-stage translator that begins with a simple basic block translator. Once hotspot code is detected, an optimizing translator is invoked. IA-32 EL is a pure software solution that reports significant startup (translation) overhead for Windows applications such as those represented by SYSmark2000. The Digital FX!32 system [5, 14] is an emulation system for running x86 Windows applications on DEC Alpha platforms. It initially uses an interpreter that discovers untranslated code which is then translated and optimized offline between program runs. DynamoRIO [4] is a software infrastructure for runtime code manipulation. It targets user-mode x86 applications on both Linux and Windows platforms.

1.3 Paper Overview

The rest of the paper is organized as follows. Section 2 provides background for the binary translation system in our proposed co-designed x86 virtual machine system. Section 3 performs analysis and modeling to identify major sources of VM runtime overhead and shows how they affect VM startup performance. Section 4 proposes techniques to reduce binary translation overhead. Preliminary evaluation of the techniques is presented in section 5. Section 6 concludes the paper.

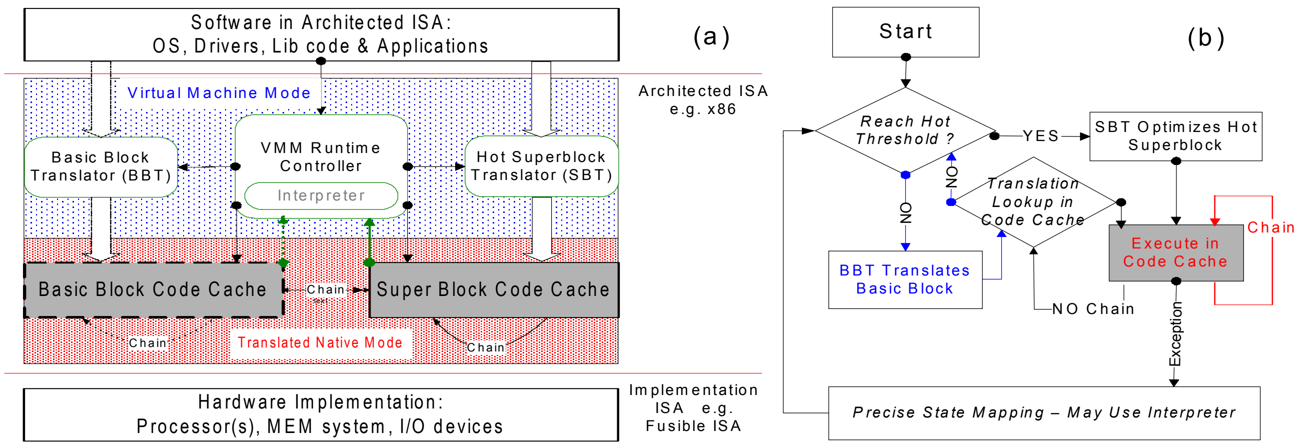
2 Baseline Co-Designed VM

To study VM startup time in a specific context, we use a co-designed x86 virtual machine based on a fusible instruction set [15,16]. The details of the design are described in a previous paper [16], and in that paper steady state performance for the SPEC2000 integer benchmarks is evaluated. In this section we briefly summarize the baseline co-designed VM.

The proposed microarchitecture and ISA cooperatively implement *macro-op execution* [16]. Pairs of dependent RISC-like micro-ops (the first micro-op generates a source operand for the second) are fused by VM software hotspot optimizer into macro-ops. Then, an enhanced and simplified superscalar microarchitecture processes the fused macro-ops as single entities throughout the *entire* pipeline. The VM hotspot optimizer can fuse micro-ops that do not belong to the same original x86 instruction (in contrast to other x86 implementations that regroup operations from the same x86 instruction for the pipeline frontend stages [9, 12, 18]). By fusing dependent pairs, the instruction issue logic (scheduler) can be efficiently pipelined, and the operand forwarding network is significantly simplified. Ignoring clock cycle advantages (which may be significant), the proposed implementation achieves an 18% IPC speedup over a conventional superscalar pipeline for the SPEC2000 integer benchmarks. Performance improvements for Windows applications are less and are given later in this section.

The VM software first decomposes (cracks) x86 instructions into micro-ops and then follows an optimization algorithm that reorders and fuses appropriate dependent micro-op pairs into macro-ops [15]. For studying startup overhead, the details of the optimization process are not important; what is important is that the optimizing DBT software has fairly high overhead (it averages over 1000 instructions to translate and optimize each x86 instruction). In addition, the hardware assists proposed in this paper target the BBT aspect of translation, not the optimizing aspects. Hence, they can be applied to other VM implementations, not just the one studied here.

The co-designed VM software consists of four major components (Fig. 1a). (1) A light-weight basic block translator (*BBT*) that generates straightforward translations for each basic block when it is first executed; (2) An optimizing hot superblock translator (*SBT*) that forms and optimizes dynamic superblocks; (3) *Code caches* -- concealed VM memory areas for holding BBT and SBT translations, and (4) the *VMM runtime system* that orchestrates the VM execution: it selects between BBT and SBT for translation, recovers precise program state, manages the code caches, etc. Fig. 1b shows the flowchart followed by VM software. When an x86 binary starts execution, the system enters the VM software (*VM mode*) and uses translations generated by BBT for initial emulation (Fig. 1b). Once a hotspot superblock is detected, it is optimized by the SBT and placed into the code cache. Branches may either be linked by the VMM runtime system initially via translation lookup table or eventually chained directly to the target translation in the code cache. For most applications, the VM software will find the working set, optimize it, and then leave the processor executing in the translated code cache as the steady state, which is defined as the *translated native mode* (shaded in Figure 1).



Software in Architected ISA: OS, Drivers, Lib code & Applications
(a)
Virtual Machine Mode
Architected ISA e.g. x86
Basic Block Translator (BBT)
VMM Runtime Controller
Interpreter
Hot Superblock Translator (SBT)
Basic Block Code Cache
Chain
Super Block Code Cache
Chain
Translated Native Mode
Chain
Hardware Implementation: Processor(s), MEM system, I/O devices
Implementation ISA e.g. Fusible ISA
Start
(b)
Reach Hot Threshold ?
YES
SBT Optimizes Hot Superblock
NO
Translation Lookup in Code Cache
NO
Chain
Execute in Code Cache
BBT Translates Basic Block
NO Chain
Exception
Precise State Mapping – May Use Interpreter

Figure 1. Staged emulation in the co-designed x86 VM

In the research reported here, we conduct experiments and evaluations on full-system Windows application traces collected from Winstone2004 Business suite [28]. These applications are more difficult to optimize than the SPEC2000 integer benchmarks. For these Windows benchmarks our baseline co-designed VM achieves 8% IPC speedup for steady state performance. The lower IPC speedup (versus SPEC2000 integer) is caused by different program characteristics between the benchmarks. The more important are: (1) only 49% of the dynamic micro-ops are fused into macro-ops versus 57% in SPEC2000 integer, and (2) larger working sets in the Winstone applications cause more cache misses that dilute IPC performance improvements. Although Winstone IPC speedups are lower than for SPEC2000 integer, they are still significant, and the co-designed VM has other advantages (clock cycle and hardware simplicity).

3 Performance Characteristics of Dynamic Binary Translation Systems

There are many performance implications caused by VM runtime translation, for example, cycles performing translation and the addition of code cache to the memory hierarchy. We first discuss performance for a translation-based VM system from a unified memory hierarchy perspective. Then, we model VM startup behavior and address some key trade-offs in a VM system featuring staged translation.

3.1 Performance Dynamics of Translation-based VM Systems

In a conventional design, when a program is to be executed, its binary is first loaded from disk into main memory. Then, the program starts execution. As it executes, instructions are moved up and down the memory hierarchy, based on usage. Instructions are eventually distributed among the levels of cache, memory, and disk.

In the co-designed VM approach, the binary containing architected ISA instructions is first loaded from disk into main memory, just as in a conventional design. However, the architected ISA instructions must be translated to implementation ISA instructions before they are executed. The translated code is held in the code cache for reuse until it is evicted to make room for other blocks of translated code. Any evicted translation must then be re-translated and re-optimized if it again becomes active. As a program executes, the translated implementation ISA instructions distribute themselves in the cache hierarchy in the same way as architected ISA instructions in a conventional system.

To simplify the analysis for co-designed VMs, especially regarding the effects of translation, we identify four primary scenarios.

(1) *Disk startup*. This scenario occurs for initial program startup or reloading tasks swapped out – the binary is loaded from disk for execution. After memory is loaded, execution proceeds according to scenario 2 below. That is, scenario 1 is the same as scenario 2 with a disk load added at the beginning.

(2) *Memory startup*. This scenario models major context switches (or program phase changes) – If a context switch is of long duration or there is a major program phase change to code that has never been executed (or has not been executed for a very long time), then the required translated code is not in the code cache. However, the architected ISA code is in main memory, and will need to be (re)translated before it can be executed. This translation time is an additional VM startup overhead which has a negative effect on performance.

(3) *Code cache startup / transient*. This scenario models the situation that occurs after a short context

switch or short duration program phase change. Translated implementation ISA code is still available in the main memory code cache, but not in the other levels of the cache hierarchy. To resume execution after the context switch (or a return to the previous program phase), there are cache misses as instructions are again fetched. However, there are no instruction translations.

(4) *Steady state*. This scenario models the situation where all the instructions in the current working set have been translated and placed properly in the cache hierarchy. The processor is running at "full" speed.

Clearly, scenario 4 *steady state* is the good case for a co-designed VM using DBT. Performance is determined mainly by processor architecture, and the co-designed VM fully achieves its intended benefits.

In scenario 3, *code cache transient*, performance is similar in both the conventional processor and VM designs as both schemes fetch instructions through the cache hierarchy, and no translation is required in the co-designed VM. Performance differences are mainly caused by second order cache effects. For example, the translated code will likely have a larger footprint in main memory, however, the code restructuring for superblock translation will lead to better temporal locality and more efficient instruction fetching.

In contrast, scenario 2 *memory startup* is the case where VM startup overhead is most exposed. The translation from architected ISA code (in memory) into implementation ISA code (in the code cache) is required and causes the biggest negative performance impact of binary translation versus a conventional superscalar design.

As noted earlier, scenario 1 *disk startup* is similar to scenario 2, with the added disk access delay. The performance effects of loading from disk are the same in both the conventional and VM systems. Moreover, the disk load time, lasting many milliseconds, will be the dominant part of this scenario. The additional startup time caused by translation will be less apparent and the relative slowdown will be much less in scenario 1 than in 2.

Based on the above analysis, we will focus our startup comparison between co-designed VMs with a baseline conventional superscalar for scenario 2. That is, when we analyze startup performance, we will start with a program binary already loaded from disk, but with the caches empty, and then track startup performance as translation and optimization is performed concurrently with execution.

Figure 2 shows a scenario 2 *memory startup* performance comparison between a conventional superscalar processor and the baseline co-designed VM that relies on software for DBT. Results are given for two staged emulation strategies for the co-designed VM; the first uses BBT followed by SBT, and the second uses interpretation followed by SBT. The specific machine configurations are given later in Table 2.

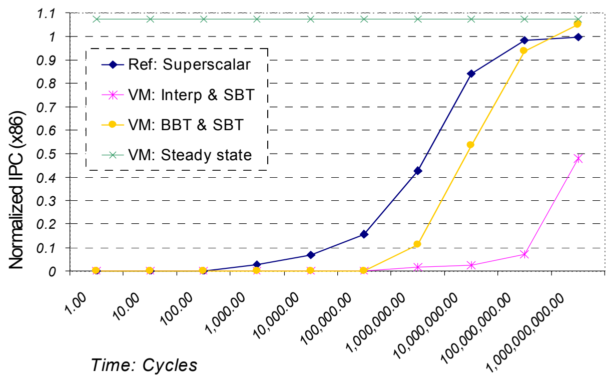


Figure 2. VM startup performance compared with a conventional x86 processor

The simulations start with empty caches and run 500-million x86-instruction traces to track performance. (The total simulation cycles range from 333-million to 923-million cycles for the reference superscalar). Results are averaged for the traces collected from the ten Windows applications in Winstone2004 Business suite. IPC performance is normalized with respect to the steady state reference superscalar IPC performance. The horizontal line across the top of the graph shows the VM steady state IPC performance gain (8%).

The x-axis shows execution time in cycles on a logarithmic scale. The y-axis shows the harmonic mean of their *aggregate* IPC, i.e. the total instructions executed up to that point divided by the total time. At a given point in time, the aggregate IPCs reflect the total numbers of instructions executed, making it easy to visualize the relative overall performance up to that point in time.

A good measure of startup overhead is the time it takes a co-designed VM to "catch up" with the reference superscalar processor. That is, the time at which the co-designed VM has executed the same number of instructions (as opposed to the time where the instantaneous IPCs are equal, which happens much earlier.) This crossover, or breakeven, point occurs later than 200-million cycles for the baseline VM system using staged BBT followed by SBT. And this co-designed VM system barely reaches half the steady-state performance gains (4%) before the traces finish.

For the co-designed VM using interpretation followed by SBT, the startup performance is much worse. A hotspot threshold for switching from interpretation to SBT is 25 executions (as derived using the method described below in Section 3.2). After finishing the 500-million instruction traces, the aggregate performance is only half that of a conventional superscalar processor.

Clearly, the runtime translation overhead affects VM startup performance when compared with a conventional superscalar processor design, especially for a startup periods less than 100-million cycles (or 50 milliseconds in a 2GHz processor core). At the one-million-cycle point, the baseline VM system has executed only one fourth the instructions of the reference superscalar scheme.

3.2 Performance Modeling of Staged Emulation

In a two-stage scheme consisting of BBT and SBT, emulation starts with simple basic block translation. Dynamic profiling is used to detect hot code regions. Once a region of code is found to be "hot", it is re-organized into superblock(s) and is optimized. Therefore, translation overhead is a function of two major items. (1) The number of static instructions touched by a dynamic program execution that need to be translated first by BBT (this number is denoted as M_{BBT}). (2) The number of static instructions that are identified as hotspot and thus are optimized by SBT (denoted as M_{SBT}). We use symbols Δ_{BBT} and Δ_{SBT} to represent per x86 instruction translation overheads for BBT and SBT, respectively. Then, for such a system, the VM translation overhead is:

$$\textit{Translation overhead} = M_{BBT} * \Delta_{BBT} + M_{SBT} * \Delta_{SBT} \quad \text{(Eq.1)}$$

Clearly, M_{BBT} is a basic characteristic of the program's execution and cannot be changed. Thus, a feasible way to reduce BBT overhead is to reduce Δ_{BBT}, and for this goal, we propose hardware assists in Section 4. Regarding the SBT overhead, we argue that good hotspot optimizations are complex and need the flexibility advantages of software. A hardware implemented optimizer (at least for the optimizations we consider) would be both complex and expensive. Fortunately, for most applications, the hotspot code is a small fraction of the total static instructions. Moreover, the hotspot size, M_{SBT}, is sensitive to the hot threshold setting. Therefore, we need to explore a balanced trade-off regarding the hot threshold setting, which not only reduces SBT overhead by detecting true hotspots, but also collects optimized hotspot performance benefits via good hotspot code coverage.

Our evaluation of this trade-off uses a specialized version of the model proposed for the Jikes virtual machine [2]. Let p be the speedup an optimized superblock can achieve over the simple basic block code. Also let N be the number of times a given instruction executes and let t_b be the per instruction execution time for code generated by BBT. Then, to break even, the following equation holds. (This assumes that the optimizer is written in optimized code and its overhead Δ_{SBT} is measured in terms of architected ISA instructions.)

$$N * t_b = (N + \Delta_{SBT}) * (t_b / p) \Rightarrow N = \Delta_{SBT} / (p - 1) \quad \text{(Eq.2)}$$

That is, the breakeven point occurs when the number of times an instruction executes is equal to the translation overhead divided by the performance improvement. In practice, at a given point in time, we do not know how many times an instruction will execute in the future. So, this equation cannot be applied in an *a priori* fashion. In the Jikes system, it is assumed that if an instruction has already been executed N times, then it will be executed at least N more, Hence, the value N as given in Equation 2 is used as the threshold value.

In our VM scheme, we set the hot threshold for triggering SBT translation based on this equation and benchmark characteristics. To calculate the hot threshold based on the equation, we first determine the parameters. For our VM system, we have measured Δ_{SBT} to be 1152 x86 instructions (say 1200) and p is 1.15 ~ 1.2, i.e. optimized SBT code runs 15 to 20 percent faster than the code generated by BBT. Then to break even, Equation 2 suggests that N should be 1200/.15 = 8000.

To illustrate our reasoning and the motivation for a relatively high hotspot threshold, we characterized the chosen Windows benchmarks. We used data averaged over the traces of length 100-million x86 instructions collected from the ten Winstone2004 Business suite applications. The x-axis of Figure 3 is instruction count (frequency). The left y-axis shows the number of static x86 instructions that are executed for the number of times marked on the x-axis. The threshold execution count (8000) is marked with a vertical line. By using the left y-axis, we see that only 3K of the static instructions have exceeded the hotspot threshold at the time 100 million instructions have been executed. It is clear from the figure that, for these benchmarks, only a small fraction of executed static instructions become hotspots.

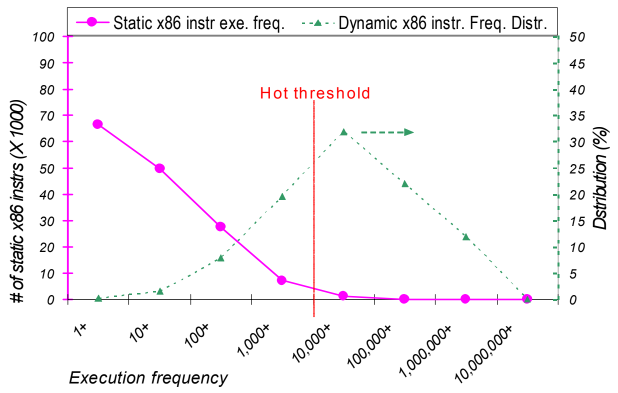


Figure 3. Winstone2004 instruction execution frequency profile (*100M x86 instruction traces*)

The right y-axis shows the distribution function of total dynamic x86 instructions. For example, the peak point of the distribution curve shows that 30+% of all dynamic instructions execute more than 10K times, but less than 100K times. This curve falls off after 100K only because the total dynamic instruction counts for the simulations are 100-million. For longer-running programs, this curve would continue to rise and the peak would shift to the right toward higher execution frequency (as the arrow shows). It is clear from the figure that for a hot threshold on the order of thousands, the hotspot coverage (the percentage of instructions executed from the optimized hotspot code) is fairly modest. However, the hotspot code coverage will be significantly improved if benchmarks run longer as in realistic cases.

Returning now to Equation 1, the average value of M_{BBT} is 150K static instructions (this can be determined by adding the data points of the static instruction curve) and the average value of M_{SBT} is 3K (determined by adding the data points of the static instruction curve that are to the right of the hot threshold line). Assuming Δ_{BBT} = 105 native instructions (as measured in our baseline VM) and Δ_{SBT} = 1674 native instructions (equivalent to the 1152 x86 instructions above), then we see that the BBT component is 105*150K = 15.75M native instructions, and the SBT component is 1674*3K = 5.02M native instructions. Therefore, in our VM system, BBT causes the major translation overhead, and this is the overhead we tackle in this paper. Also because it is a simpler operation, BBT offers more opportunities for hardware assists.

4 Hardware Assists for Binary Translation

We now propose techniques that significantly reduce VM startup overhead. These new hardware mechanisms accelerate certain computation intensive parts of the translation process. However, we continue to employ VMM software to manage the overall translation process, due to its flexibility and simplicity. We present two hardware assists: one is placed in the pipeline frontend, and the other in the backend.

4.1 Frontend Dual Mode Decoders

In most conventional designs, x86 instructions are decoded into RISC-style operations called *micro-ops (μops)*. We leverage this approach by proposing a dual mode (two-level) decoder (Fig.4) that targets CISC ISAs. The two-level decoder is similar to the microcode engine used in the Motorola 68000 [26]. The first level decoder cracks x86 instructions into "vertical" micro-ops – in the same 16-bit/32-bit micro-op format that is used in the baseline co-designed VM implementation ISA [16]. Then, a second level decoder generates the "horizontal" decoded control signals used by the pipeline.

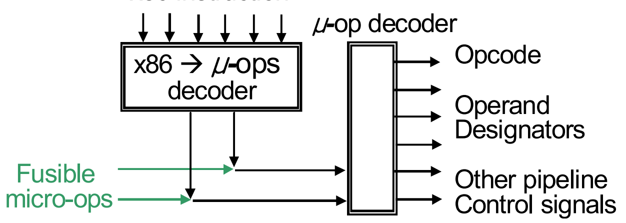


Figure 4. Dual mode x86 decoder

A two-level decoder is especially well suited to a CISC ISA because complex CISC instructions must both be decomposed (*cracked*) into RISC-style micro-ops and decoded into pipeline control signals.

We then add a direct path into the second level decoder (Fig.4), which enables the decoder to be used in two modes. In conventional *x86-mode*, x86 instructions are fetched from memory, and both decode levels are used. In translated *native-mode*, translated implementation instructions are fetched from the code cache. These instructions bypass the first level decoder and only use the second level decoder. With the dual-mode decoders, both architected ISA (x86) code and implementation ISA instructions can be processed by the pipeline. The ability to support x86 mode eliminates the need for BBT, along with its translation overhead and any side effects on the memory hierarchy.

As the processor runs, it switches back and forth between x86-mode and native-mode, under the control of VMM software. When executing in x86-mode, the x86 instructions pass through both decode levels (Fig. 5); this is done when a program starts up, for example. In x86-mode, performance will be similar to a conventional x86 superscalar implementation.

A side effect of using the dual-mode approach is that profiling software cannot be embedded into BBT code, because there is no BBT code. As a consequence, the design should employ profiling hardware similar to that used by Merten *et al.* [23]. This hardware's sole function is to detect hotspots. When such a hotspot is detected, the hardware invokes the VMM software which can then organize hotspot code into superblock(s), translate, and optimize it, and place the optimized superblock(s) into the code cache.

Dual mode decoders are fast and fit well in a conventional superscalar design. The replacement of a single-level decode table with a two-level decoder may be a good hardware tradeoff which will likely result in fewer transistors than a single-level design, as explained by the Motorola 68000 designers [26]. This approach, extended to dual mode operation, adds relatively little extra hardware to a conventional implementation -- the bypass path around the first level decoder. Also, when executing the optimized hotspot code, the first level decoder is bypassed and can be powered off.

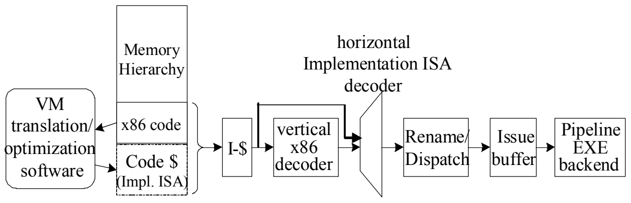


Figure 5. Dual mode x86 decoders in a pipeline

4.2 Backend Functional Units

The frontend hardware assist modifies a critical part of the pipeline and must be able to decode instructions at full bandwidth. Furthermore, it must be designed to implement the complete architected ISA (x86). An alternative is to implement a hardware assist in the form of a programmable backend functional unit. This unit is less intrusive than the frontend unit, does not need to provide the high bandwidth of the frontend assist, and can target the common cases, not all cases.

During initial emulation, BBT introduces the major runtime overhead, and the dominant part of BBT is to decode and crack x86 instructions into micro-ops. In our BBT system, an average of 90 out of the 105 micro-ops used for translating each x86 instruction are associated with instruction decoding and cracking. Therefore, a functional unit that performs these operations can greatly speed up BBT translation.

We propose such a backend functional unit that is accessed through a new instruction in the implementation ISA. Table 1 briefly describes the new instruction XLTx86. XLTx86 accesses the 128-bit F registers that are architected for mapping the x86 FP/media states. Additionally, XLTx86 operates on a special flag/status register CSR that is explained below.

Table 1: Hardware accelerator -- new instruction

NEW INSTRUCTION: XLTX86 FSRC, FDST
BRIEF DESCRIPTION: Decode an x86 instruction aligned at the beginning of the 128-bit Fsrc register, and generate 16b/32b micro-ops into the Fdst register. This instruction affects CSR status register

Figure 6a illustrates the kernel loop used by the VMM for hardware accelerated BBT (in the implementation ISA assembly language). Rx86pc is the implementation register holding the architected x86 PC value; this register points to an instruction in the x86 instruction memory.

To be more specific, x86 instructions are fetched by a LD (load) operation into register Fsrc. Because x86 instructions are from one byte to seventeen bytes long (and very few are more than eleven bytes in real code), the Fsrc register holds at least one x86-instruction. The fetched x86 instruction is aligned at the beginning of the Fsrc register. The next instruction, XLTx86, then decodes and cracks the x86-instruction into micro-op(s). The input to XLTx86 is the Fsrc register. The output micro-ops are placed in the Fdst register and flags are set in the CSR status register. The format of the flag status register CSR is shown in Figure 6b. The 4-bit *x86_ilen* field returns the length of the x86 instruction. The 4-bit *μops_bytes* field returns the length of the generated micro-op(s) in the implementation ISA. The *Flag_cmplx* bit is set if the x86-instruction being decoded is too complex for the hardware decoder. This mechanism keeps the hardware assist simple and fast by off-loading complicated cases to software; for example, if the x86 instruction should happen to be more than 16 bytes (the size of the Fsrc register). The *Flag_cti* flag bit is set if the x86-instruction being processed is a control transfer instruction (which requires special handling). After decoding, most x86-instructions are cracked into micro-ops of no more than 16 bytes. Note that the 16-bit/32-bit implementation ISA design implies that, only in a few rare cases, the 128b Fdst is too short to hold result micro-ops; this is another case that is flagged as a complex instruction. Native micro-ops in Fdst are written back to the code cache by a store operation. The rest of the loop does bookkeeping (in the figure, :: indicates the two micro-ops are fused as a macro-op [15.16]).

```
0.  HAloop:
1.  LD       Fsrc, [Rx86pc]
2.  XLTx86   Fdst,  Fsrc
3.  Jcpx     complex_x86code
4.  Jcti     branch_handler
5.  ST       Fdst, [Rcode$]
6.  MOV      Rt0, CSR
7.  AND      Rt1, Rt0, 0x0f :: ADD   Rx86pc, Rt1
8.  AND.x    Rt2, Rt0, 0xf0 :: ADD   Rcode$, Rt2
9.  JMP      HAloop
```

(a). Code for the HW assisted fast BBT loop

Flag_cti	*Flag_cmplx*	*μops_bytes (4-bit)*	*x86_ilen (4-bit)*

(b). CSR, control & status register format for XLTx86

Figure 6. HW accelerated basic block translator

For microarchitecture design, the new functional unit is located in the FP/media part of the processor core because it uses F registers to hold long x86 instructions and multiple micro-ops (see Figure 7). If implemented in a superscalar style microarchitecture such as that proposed in [16], the XLTx86 instruction will be dispatched to the FP/media instruction queue(s) and issued to the new functional unit via a FP/media issue port. XLTx86 can take multiple cycles to execute as do many other FP/media instructions. In our research, we assume XLTx86 takes four cycles. The x86 instruction bytes are supplied to the functional unit via streaming buffer and the generated micro-ops are written back to memory directly without going through the data cache.

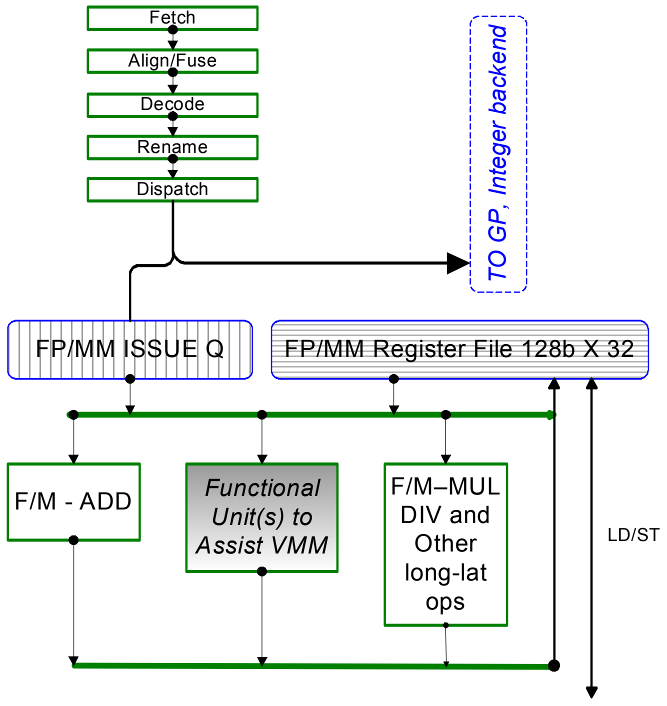

Fetch
Align/Fuse
Decode
Rename
Dispatch
TO GP, Integer backend
FP/MM ISSUE Q
FP/MM Register File 128b X 32
F/M - ADD
Functional Unit(s) to Assist VMM
F/M–MUL DIV and Other long-lat ops
LD/ST

Figure 7. Hardware Accelerator *μ*-architecture

For circuit design, the functional unit for XLTx86 is essentially a simplified, one instruction wide, x86 decoder relocated to the execution stage of the FP / media core.

The new instruction, XLTx86, speeds up BBT by accelerating the dominant part of fetch/decode/crack (from tens of cycles to only a few cycles). Meanwhile, because it is an instruction that provides a primitive operation (from the translation perspective), it offers the VMM flexibility and simplicity beyond the frontend dual mode decoders.

5 Evaluation

5.1 Evaluation Methodology

The experimental infrastructure is based on the co-designed x86 virtual machine [16] summarized in Section 2. The staged translation software is developed as part of the concealed VMM runtime software. The co-designed processor is modeled via a microarchitecture timing simulator.

To compare startup performance with conventional superscalar designs and to illustrate how VM system startup performance can be improved by the proposed hardware assists, we simulate the following machine configurations. Detailed configuration settings are provided in Table 2.

- ***Ref*: *superscalar:*** The conventional superscalar microarchitecture serves as the baseline/reference.
- ***VM.soft***: A conventional co-designed VM scheme, with software-only BBT and SBT dynamic binary translators.
- ***VM.be***: The co-designed x86 VM, equipped with pipeline backend functional units.
- ***VM.fe***: The co-designed x86 VM, equipped with dual mode decoders at the pipeline frontend.

The hot threshold in VM systems is determined by Equation 2 and benchmark characteristics. For the benchmarks used in this paper, all VM models *VM.soft*, *VM.be* and *VM.fe*, use a threshold of 8K. Note that the *VM.fe* and the *Ref: superscalar* schemes have a longer pipeline frontend due to the x86 decoders.

To stress startup performance and other transient phases for a binary translation based VM system, we run short traces collected from the ten Windows applications taken from the Winstone2004 Business benchmarks. For studies focused on accumulated values such as benchmark characteristics, we simulate 100-million x86 instructions. For studies focused on time variations, such as variation in IPC over time, we simulate 500-million x86 instructions and express time on a logarithmic scale. All simulations are set up for testing the memory *startup* scenario (Scenario 2 described in Section 3.1) to stress VM specific runtime overhead.

We first show how the proposed hardware assists speed up VM runtime translation by comparing the VM system startup performance with that of a conventional superscalar processor model. Then, we conduct performance and energy analysis for the hardware assists integrated into the VM system.

5.2 Performance Evaluation of the VM Systems

Figure 8 shows the same startup performance comparisons as Figure 2. Additionally Figure 8 shows startup performance for the VMs containing the proposed hardware assists. As before, the normalized IPC (harmonic mean) for the VM steady state is about 8% higher than the baseline superscalar when it is in steady state.

The VM system equipped with dual mode decoders at the pipeline frontend (*VM.fe*) shows practically a zero startup overhead; performance follows virtually the same startup curve as the baseline superscalar because they have very similar pipelines for cold code execution. Once a hotspot is detected and optimized, the VM scheme starts to reap performance benefits. *VM.fe* reaches half the steady-state performance gains (4%) in 100M cycles.

The VM scheme equipped with a backend functional unit decoder (*VM.be*) also demonstrates good startup performance. However, compared with the baseline superscalar, *VM.be* lags behind for the initial several millions of cycles. The breakeven point occurs at around 10-million cycles and the half performance gain point happens after 100-million cycles. After that, *VM.be* performs very similarly to the *VM.fe* scheme.

Table 2. Machine Configurations

	Ref: superscalar	VM.soft	VM.be	VM.fe
Cold x86 code	Hardware x86 decoders No optimization	Simple software BBT, no opts	BBT assisted by the backend HW decoder.	Hardware Dual-mode decoders
Hotspot x86 code	Hardware x86 decoders No optimization	Software hotspot optimizations	Perform the same hotspot optimization as in VM.soft, with simple HW assists.	
ROB, Issue buffer	36 issue queue slots, 128 ROB entries, 32 LD queue slots, 20 ST queue slots			
Physical Register File	128 entries, 8 Read ports, 5 Write ports	128 entries, 8 Read and 8 Write ports (2 Read & 2 Write ports are reserved for the 2 memory ports).		
Pipeline width	16B fetch width, 3-wide decode, rename, issue and retire.			
Cache Hierarchy	L1 I-cache: 64KB, 2-way, 64B lines, Latency: 2 cycles. L1 D-cache, 64KB, 8-way, 64B lines, Latency: 3 cycles. L2 cache: 2MB, 8-way, 64B lines, Latency: 12 cycles			
Memory Latency	Main memory latency: 168 CPU-cycles. 1 memory cycle is 8 CPU core cycles.			

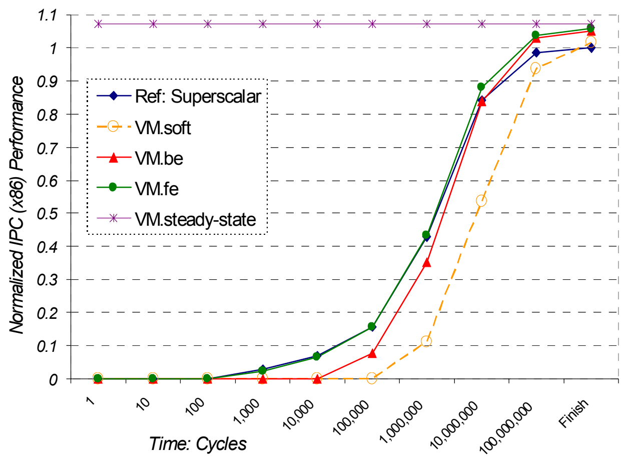


Figure 8. Startup performance comparison

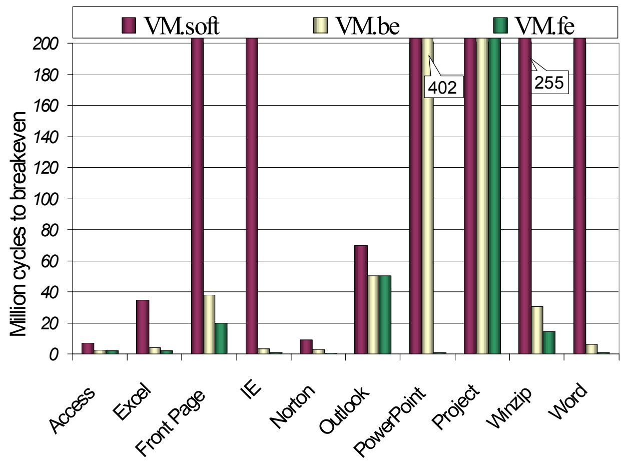


Figure 9. Breakeven points for individual traces

Figure 9 shows, for each individual benchmark, the number of cycles a particular translation scheme needs to first reach the breakeven point with the reference superscalar. We label bars that are higher than 200-million cycles (to break even) with their actual values. Otherwise, a bar that is higher than 200-million cycles means its VM model did not break even within the 500-million x86-instruction trace simulation.

It is clear from the figure that, in most cases, using either the frontend or the backend assists can significantly reduce the VM startup overhead and enable VM schemes to break even with the reference superscalar within 50-million cycles. However, for the *Project* benchmark, the VM schemes cannot break even within the tested runs, though they do follow performance of the reference superscalar closely (within 5%). Further investigation indicates that the VM steady state performance for Project is only 3% better than the superscalar baseline, thus the VM schemes take a longer time to collect enough hotspot performance gains to compensate for the performance loss due to initial emulation and translation.

5.3 Performance and Energy Analysis

It is straightforward to explain the startup performance improvement for *VM.fe* because its x86-mode execution is very similar to the execution of a baseline superscalar. On the other hand, the *VM.be* scheme translates cold code in a co-designed way that still involves VM software. Consequently, we consider how VM software overhead is reduced after being assisted by the XLTx86 instruction. The software-only baseline VM (*VM.soft*) is measured to spend on average 9.9% of its runtime performing BBT translation, for the first 100-million dynamic x86 instructions.

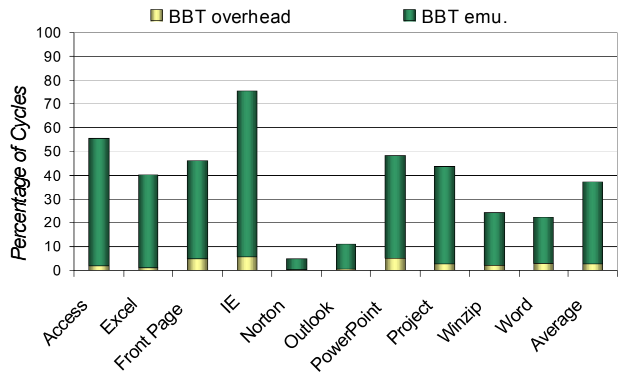


Figure 10. BBT translation overhead & emulation cycle time (100M x86 instruction traces)

Figure 10 shows how VM cycles (for the *VM.be* scheme) are spent. For each benchmark, the lower bars (BBT overhead) represent the percentage of VM cycles spent for BBT translation and the upper bars (BBT emu.) indicate the percentage of cycles the *VM.be* model executes basic block translation. The rest of the cycles are mostly spent for SBT translation and emulation with the optimized hotspot code. To stress startup overhead, the data is collected for the first 100M x86 instructions for each benchmark.

It is evident from Figure 10 that after adding the new XLTx86 instruction at the pipeline backend, the average BBT translation overhead is reduced to 2.7%; about 5% at worst. Further measurements indicate that the software-only BBT spends 83 cycles to translate each x86-instruction (including all BBT overhead, such as chaining and searching the translation lookup table). In contrast, *VM.be* needs only 20 cycles to do the same operations.

After BBT translation, the *VM.be* scheme spends 35% of its total cycles (*BBT.emu* bars in Fig. 10) executing BBT translations. The execution of BBT translations is less efficient than that of SBT translations. However, this BBT emulation does not lose much performance because the BBT translations run fairly efficiently (On average 82~85% IPC performance of SBT optimized code). This IPC performance is only slightly less than the baseline superscalar design. And for program startup transient, cache misses dilute CPU IPC performance.

The rest of the *VM.be* cycles (*VM.fe* is similar) are spent in SBT translation (3.2%) and emulation via SBT translations (59%). The optimized SBT translations improve overall performance by covering 63% of the 100M x86 instructions. For 500-million x86 instructions runs, the hotspot coverage rises to 75+% on average and is projected to be higher for full program runs.

A software-based co-designed VM does not require complex x86 decoders in the pipeline as in conventional x86 processors. This can provide significant energy savings (one of the motivations for the Transmeta designs). However, when hardware x86 decoder(s) are added as assists, they consume energy. Nevertheless, this energy consumption can be mitigated by powering off the hardware assists when they are not in use.

To estimate the energy consumption, we measure the activity of the hardware x86 decoding logic. The activity is defined as the percentage of cycles the decoding logic needs to be turned on. Figure 11 shows the activity for the four machine configurations. The x-axis shows the cycle time on logarithmic scale and the y-axis shows the *aggregate* decoding logic activity.

For conventional x86 processors, x86 decoders are always on (except Pentium 4 [13]). In contrast, for the *VM.be* scheme, the hardware assist activity quickly decreases after the first 10,000 cycles. It becomes negligible after 100-million cycles. Considering that only one decoder is needed to implement XLTx86 in the *VM.be* scheme, energy consumption due to x86 decoding is mitigated. For the *VM.fe* model, the dual mode decoders at the pipeline frontend need to be active if the VM is not executing optimized hotspot code. The decoders' activity also decreases quickly, but later than a *VM.be* scheme as illustrated in the figure.

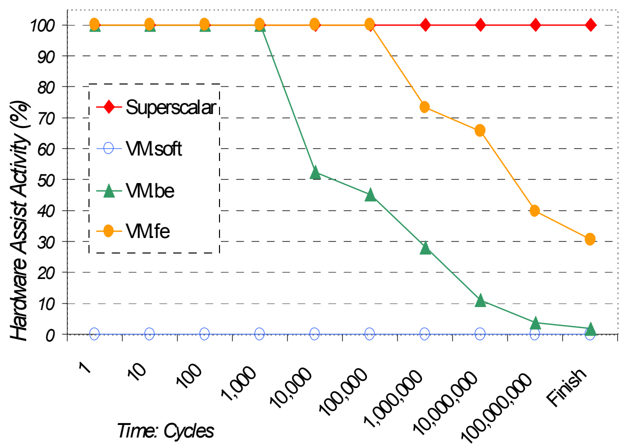


Figure 11. Activity of HW assists

6 Conclusions and Future Directions

To understand runtime overhead in co-designed VMs, we investigated a co-designed x86 VM using fused micro-ops. Runtime overhead can be caused by emulation of non-hotspot code. This is especially a problem for the CISC x86 whether using an interpreter or BBT. Dynamic hotspot optimization overhead can also be detrimental to startup performance. Runtime overhead not only affects startup performance, but also system performance consistency and predictability.

In our exploration of VM startup performance, we first reduce baseline VM translation overhead by employing a staged translation strategy and simple, but efficient, BBT translation as in many current VM systems. Then, we analyze and model VM startup performance from a memory hierarchy perspective. We observe that non-hotspot emulation causes the major translation overhead and hotspot optimization can further exacerbate the VM startup curve for some applications. We propose an overall strategy to reduce VM startup time. To reduce translation overhead for non-hotspot code emulation, we propose hardware assists. These assists significantly reduce (or eliminate) BBT runtime overhead.

After applying the strategy and integrating the accelerated binary translators into the baseline co-designed x86 virtual machine, we show that the VM system startup performance is significantly improved for the Windows application benchmarks. Considering the fact the virtual machine design enables a novel efficient microarchitecture, the overall system performance is improved without sacrificing design complexity.

For future research, we anticipate that the combination of the adaptive translation strategies with simple hardware accelerators is not limited to the co-designed virtual machine paradigm. The ideas can be applied to speed up other dynamic binary translation systems, thus enabling other attractive system features and capabilities.

Acknowledgements

We appreciate the contributions of Mikko Lipasti and Ilhyun Kim regarding fused architectures and Wooseok Chang's help with Windows workloads and trace collection. This work was supported by NSF CCR-0311361 and the Intel Corporation.

References

1 E. R. Altman, *et al.*, "Advances and Future Challenges in Binary Translation and Optimization", *Proc. of the IEEE, Special Issue on Microprocessor Architecture and Compiler Technology*, pp. 1710-1722, Nov. 2001.

2 M. Arnold, *et al.*, *"*Adaptive Optimization in the Jalapeño JVM*" ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '00)*, pp. 47-65, Oct. 2000.

3 L. Baraz, *et al.* "IA-32 Execution Layer: a two phase dynamic translator designed to support IA-32 applications on Itanium®-based systems", *Proc. of the 36th Int'l Symp. on Microarchitecture,*, pp. 191-204, Dec. 2003.

4 D. Bruening, *et al.,* "An infrastructure for adaptive dynamic optimization". *Proc. of the 1st Int'l Symp. on Code Generation and Optimization*, pp. 265-275, March 2003.

5 A. Chernoff, *et al*, "FX!32: A Profiler-Directed Binary Translator*", IEEE Micro (18)*, March/April 1998.

6 Y. Chou, J. P. Shen. "Instruction Path Coprocessors", *Proc. of the 27th Int'l Symp. on Computer Architecture*, pp. 270-281, June 2000.

7 J. Dean, *et al*. "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors", *Proc. of the 27th Int'l Symp. on Computer Architecture*, pp. 316-325, Jun. 2000.

8 J. C. Dehnert, *et al.* "The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges", *Proc. of the 1st Int'l Symp. on Code Generation and Optimizations*, pp. 15-24, Mar. 2003.

9 K. Diefendorff "K7 Challenges Intel" *Microprocessor Report*. Vol.12, No. 14, Oct. 25, 1998

10 K. Ebcioglu, E.. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility", *Proc. of the 24th Int'l Symp. on Computer Architecture*, pp.26-37, Jun. 1997.

11 K. Ebcioglu *et al.*, "Dynamic Binary Translation and Optimization", *IEEE Transactions on Computers*, Vol. 50, No. 6, pp. 529-548. June 2001.

12 S. Gochamn *et al.*, "The Intel Pentium M Processor: Microarchitecture and Performance", *Intel Technology Journal*, vol7, issue 2, pp. 21-36, 2003.

13 G. Hinton *et al.* "The Microarchitecture of the Pentium 4 Processor", *Intel Technology Journal*. Q1, 2001.

14 R. J. Hookway, M. A. Herdeg, "Digital FX!32: Combining Emulation and Binary Translation", *Digital Technical Journal*, vol. 9, No. 1, Jan. 1997.

15 S. Hu, J. E. Smith, "Using Dynamic Binary Translation to Fuse Dependent Instructions", *Proc. of the 2nd Int'l Symp. on Code Generation and Optimization*, pp. 213-224, Mar. 2004.

16 S. Hu, *et al.*, "An Approach for Implementing Efficient Superscalar CISC Processors", *Proc. of the 12th Int'l Symp. on High Performance Computer Architecture*, pp. 40-51, Feb. 2006.

17 W. W. Hwu *et al.*, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation", *The Journal of Supercomputing*, 7(1-2), pp. 229-248, 1993.

18 C. N. Keltcher, *et al.*, "The AMD Opteron Processor for Multiprocessor Servers", *IEEE MICRO*, pp. 66-76, Mar.-Apr. 2003.

19 H.-S. Kim, J. E. Smith, "Dynamic Binary Translation for Accumulator-Oriented Architectures", *Proc. of the 1st Int'l Symp. on Code Generation and Optimization*, pp. 25-35, Mar. 2003.

20 H.-S. Kim, J. E. Smith, "Hardware Support for Control Transfers in Code Cache". *Proc. of the 36th Int'l Symp. on Microarchitecture* pp. 253-264, Dec. 2003

21 A. Klaiber, "The Technology Behind Crusoe Processors", *Transmeta Technical Brief*, 2000.

22 K. Krewell, "Transmeta Gets More Efficeon" *Microprocessor report*. v.17, October 2003

23 M. C. Merten, *et al.* "An Architectural Framework for Runtime Optimization", *IEEE transactions on Computers*, Vol. 50, No.6, pp. 567-589, Jun. 2001.

24 S. J. Patel, S. S. Lumetta, "rePLay: a hardware framework for dynamic optimization", *IEEE, Transactions on Computers*, pp. 590-680, Jun. 2001.

25 R. Rosner, *et al.* "Power Awareness through Selective Dynamically Optimized Traces", *Proc. of the 31st Int'l Symp. on Computer Architecture*, pp. 162-175, Jun. 2004.

26 E. P. Stritter, *et al.*, "Microprogrammed Implementation of a Single Chip Microprocessor", *Proc. of the 11th Annual Microprogramming Workshop*, pp. 8-16, Nov. 1978.

27 Transmeta Corporation. Transmeta Efficeon Processor, *http://www.transmeta.com/efficeon*

28 VeriTest, PC Magazine, "Business WinStone Benchmark", *http://www.veritest.com/benchmarks/bwinstone/*