Saving and Restoring Implementation Contexts with co-Designed Virtual Machines

Ashutosh S. Dhodapkar and James E. Smith

Dept. of Electrical and Computer Engineering
University of Wisconsin-Madison
{dhodapka. jes}@ece.wisc.edu

Abstract

It is proposed that a co-designed Virtual Machine Monitor (VMM) can be used for saving and restoring implementation state when context switches occur in a manner that is completely transparent to all conventional software. As an extended case study, we use the VMM to save and restore branch predictor contents. For an 8kbit *gshare* predictor and a context switch interval of 100K instructions, the performance benefit is 4% over initializing the predictor to weakly taken following context switches (the best alternative we could find).

1 Introduction

For many years a dominant trend in processor microarchitecture has been toward increasing amounts of dynamic adaptation based on program behavior. A very early example is cache memory: the instructions or data held in a cache reflect dynamic program behavior and change as the program changes. Other examples are branch prediction and prefetch algorithms based on prior program behavior [1]. More recently, dynamically configurable hardware has been proposed for enhancing performance or power efficiency [2-5]. These latter schemes typically record dynamic program performance information in order to guide the hardware configuration. Virtually all the dynamic methods "learn" program behavior by maintaining tables of *implementation state*, and in some cases the total amount of implementation state may be quite large. The implementation state, by definition, is not specified as part of the architecture and is not needed for correct program operation. Whenever there is a program context switch much of this implementation state (and learned program behavior) may be lost and must be re-learned when the program resumes execution.

Our overall research is targeted at co-designed virtual machines where hidden, implementation-specific software is designed in conjunction with the hardware and is used to optimize performance and/or power. Virtual machine software provides an important means for shifting complexity from hardware to software. It can perform a number of functions, including binary translation and optimization [6-7], implementation of complex instructions [8], and management of configurable hardware. We propose that an important additional function is saving and restoring implementation state when context switches occur.

This incurs some time overhead, but it is more than offset by substantially reduced learning times required by adaptive performance techniques.

In following sections, we describe a simple microarchitecture that underlies a co-designed virtual machine, and illustrate the ability to save/restore implementation state by using a branch predictor as a detailed case study.

2 Co-Designed Virtual Machines

The base technology is co-designed virtual machines. Here, the processor implementers have access to a region of memory completely hidden from conventional software. The hidden memory is a portion of real memory that is concealed from the conventional software at boot time and is used only by the virtual machine. Virtual machine software is stored in ROM and copied to a portion of hidden memory as the first step of the boot sequence. The remainder of hidden memory is used as a VMM-managed software cache for translated code.

The VMM, co-designed with the hardware, provides implementers with an important mechanism for shifting complexity from hardware to software. The two best-known systems using this approach are the Transmeta Crusoe [6] and IBM Daisy [7], where the primary function is code translation and dynamic optimization. In these systems, a binary translator/optimizer converts instructions from a virtual instruction set (e.g. x86 or PowerPC) onto an implementation instruction set (VLIW for both Crusoe and Daisy). A virtual machine monitor (VMM) manages translation and optimization functions.

Because the Crusoe and Daisy processors both translate conventional instruction sets onto VLIW architectures, binary translation is a major portion of their VM implementations. In general, however, one could use a more conventional superscalar implementation, with less aggressive translation and optimization, and/or one could perform no binary translation at all, and use the VMM only for managing hardware resources (Fig. 1). In this latter scheme, the virtual instruction set and implementation set are essentially the same. However, the implementation instruction set may have a few additional instructions to allow it to interact directly with the hardware implementation. As shown in the figure, the VMM can use these instructions to read hardware monitoring information (for example, branch prediction accuracies), and write hardware configuration information. Also, the VMM can be given the capability of reading and writing implementation state. This information can be used for collecting additional dynamic program information or for putting software into the dynamic optimization loop (for example with a software-directed trace optimization [9]). This feature can also be used for saving and restoring parts of the implementation state, as we consider in this paper.

2.1 Microarchitecture

The microarchitecture of a VMM implementation is in Fig. 2. In the figure, co-designed VM hardware has been integrated into a generic pipeline. The pipeline consists of a program counter, the instruction cache, decode logic, and the remainder of the pipeline(s) simply labeled as such. The added VM logic is primarily located around the MUX that feeds the PC. This MUX normally has inputs for performing the PC increment, branches, vectored interrupts, etc. To this MUX has been added two inputs, one for a VMM entry point (the place in hidden memory where the VMM starts executing) and for restoring the original PC after VM software is finished executing. The VMM takes control of the processor by saving the current PC value in a special PC save register and switching the MUX to select the VMM entry point.

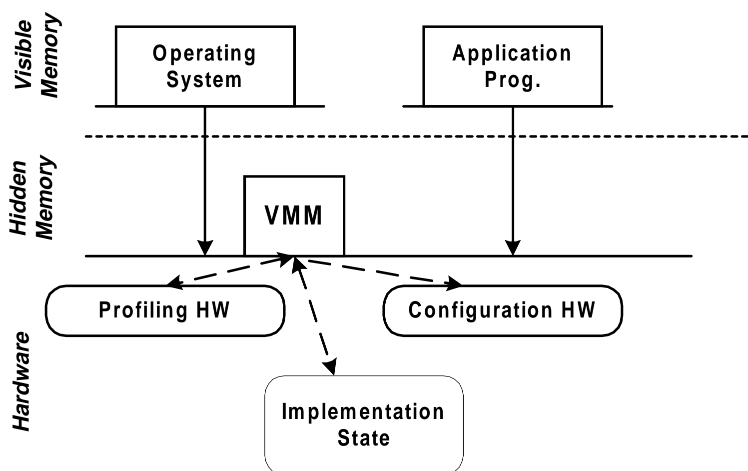

Figure 1: A Co-Designed Virtual Machine. Conventional software is in visible memory and the Virtual Machine Monitor (VMM) is in hidden memory. The VMM can access profiling hardware, adjust configuration hardware, and read/write implementation state. In the Transmeta Crusoe and IBM Daisy, the VMM also manages binary translation (not shown in the figure).

The VMM can take control in any number of situations determined by the designers, as illustrated in the figure. For example, performance monitor conditions can initiate transfers to the VMM, as can temperature sensor values. The VMM can also manage an implementation timer to take control after a period of time that it selects. Certain opcodes can cause the VMM to be entered, as can trap conditions. The VMM can save any or all of the processor's register context into hidden memory and/or it can use its own dedicated implementation registers. Then, VMM software can read performance and temperature information, and reconfigure the hardware if needed. In effect, the VMM is a micro-operating system developed specifically to manage the implementation. It is completely hidden from conventional software (just as binary translation is completely hidden in Crusoe and Daisy).

2.2 Saving and Restoring Implementation Context

As stated in the introduction, many of the techniques for dynamic performance enhancement use a large amount of implementation state that reflects the history of the current program. It takes some time to accumulate this information, and whenever there is a context switch, some or all of this information may be lost. Consequently, performance is reduced when the program eventually resumes execution. This can be avoided, however, if the VMM can read the implementation state and save it to a data area in hidden memory. Then, when the program context is restored, the implementation context can also be restored.

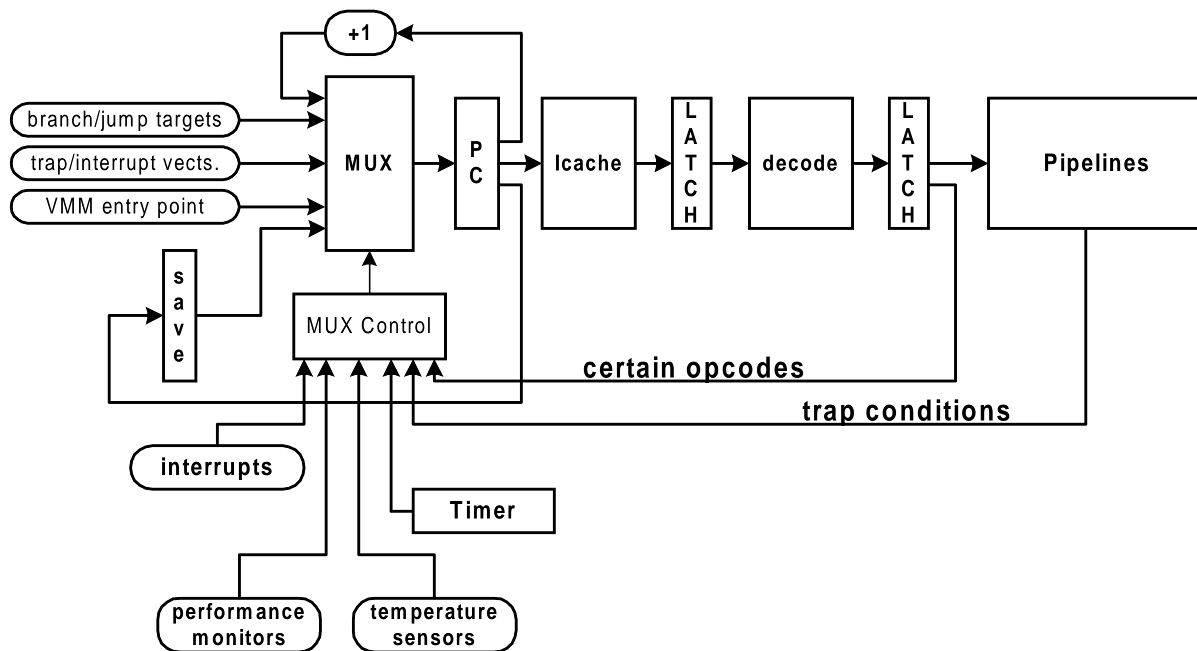

Figure 2 Co-Designed VM microarchitecture.

An important issue is the ability to identify the implementation contexts without relying directly on conventional software. This can be done in at least two ways. One way is to rely on architected control registers that contain context-specific information. For example, different contexts will typically use different page tables, so an architected control register that points to a process's page table can often be used to distinguish the context. Another approach is for the VMM to hash the PC and some of the architected register values to provide a context identifier at the time a context is switched out. Then, this hash value can be used by the VMM as a way of identifying saved implementation contexts when a new context is loaded. A key point is that the implementation state does not have to be correct for correct architected operation. That is, if the VMM occasionally loads the wrong implementation context the worst thing that can happen is performance loss.

As an extended example, we consider the implementation state contained in branch prediction tables. We assume the branch predictor table contents can be read and written via special implementation-dependent instructions [10] available to the VMM. Whenever the operating system is about to switch to a new context, the VMM can save the branch predictor contents, and load the predictor contents for the new context.

3 Context Switches and Conditional Branch Prediction

Previously, researchers have observed that context switches can affect branch prediction accuracy [11-13]. Many real systems execute a multi-programmed workload and may experience a large number of context switches. Context switch frequency depends on a number of factors such as the number of applications active on the system, the types of applications, and OS scheduling. For example, we performed some simple measurements on a 800MHz Pentium III based workstation running Linux, and the context switch frequency varied from around 300/sec to 5000/sec depending on the load and types of applications running. If we assume one instruction is executed per clock cycle, the higher end corresponds to a context switch every 150K instructions. Keeping these numbers in mind, we conducted our experiments over context switch intervals ranging from 100K to 1000K instructions. These numbers are also in line with other studies that consider the effect of context switches on branch prediction [11-13].

3.1 Estimating Performance Losses

First we estimate the predictor-related performance losses due to context switches. For this study, we model a processor along lines of a modern superscalar processor having an 8Kbit gshare predictor [14] and a 10-bit global history register. A more complete description of the processor configuration is shown in Table 1. All simulations were performed on a modified version of SimpleScalar 3.0b [15] running Alpha ISA binaries. We used integer benchmarks from the SPEC 2000 suite, and chose Gcc, Eon, Gzip, Perl, Gap and Mcf in order to cover a wide range of predictor performance. The branch prediction miss rates for these benchmarks ranged from 0.9% to 14%. All the simulations were run for 400 million instructions.

We first evaluated branch predictor performance with no context switches, and then estimated the performance loss with context switches at intervals of 100K, 200K, 500K and 1000K instructions. At each context switch interval, the predictor is re-initialized to model the effect of the context switch. The following methods were used for initializing the predictor after a context switch.

1) Random – The branch predictor contents are initialized to random values.
2) "Worst Case" (or at least an extremely bad case) – The predictor entries are reversed; that is, if a predictor entry was either strongly or weakly not taken prior to the context switch, it is initialized to strongly taken after the switch. Similarly the taken branch entries are re-initialized to strongly not taken.

3) Fixed – All the predictor entries are initialized to the same fixed value after a context switch. There are four sub-cases: strongly not taken, weakly not taken, weakly taken, and strongly taken.

Parameter	Value
RUU size	64 entries
LSQ size	32 entries
Fetch queue size	4 instructions/cycle
Decode, Issue, Commit width	4 instructions/cycle
Branch Predictor	ghsare: 4K entries, 10-bit GHR
Return Address Stack	32 entry
Branch Target Buffer	1K entry, 4 way
Functional Units	4 Integer ALUs
	1 Integer Multiply/divide
	4 Floating point ALUs
	1 Floating point Multiply/divide
L1 Instruction and Data caches	1K sets, 2-way, 32 byte block size
L2 Unified cache	2K sets, 4-way, 64 byte block size

Table 1: Processor configuration used in simulations.

Predictor performance was measured using the number of mispredictions per 1000 instructions (Fig. 3), and overall processor performance is measured in clock cycles per 1000 instructions (Fig. 4). The baseline case is *nada* -- the one with no context switches.

When the predictor state is randomized at a context switch point, there are approximately 300-600 additional mispredictions as the predictor re-learns the branch history. This is reflected in Fig. 3 where there are 3-6 additional mispredictions per 1K instructions with a context switch interval of 100K instructions. Because the branch misprediction penalty is approximately 9 cycles on average, the performance loss is on the order of 30-50 cycles per 1K instructions as shown in Fig. 4. For larger context switch intervals, the total number of additional mispredictions does not change much, but because of the larger context switch intervals, the average per 1K instructions diminishes. At 1000K intervals, the number of additional mispredictions per 1K instructions is slightly less than 1, and the performance loss is less than 10 cycles per 1K instructions. Percentage-wise, the performance degradation for randomizing predictor state was as high as 11%, with and average of 8.4% for context switch intervals of 100K instructions. For 1000K instruction intervals, the maximum degradation was 2% while the average was 1.5%.

As expected, the performance loss for the "worst case" initialization was very bad. The maximum performance loss ranged from around 18% for intervals of 100K instructions to 4.7% for 1000k instructions, while averages varied from 13.8% to 3.3%. In practice, there could be pathological cases, due to a complete mismatch of two contexts, which could lead to performance degradations this severe.

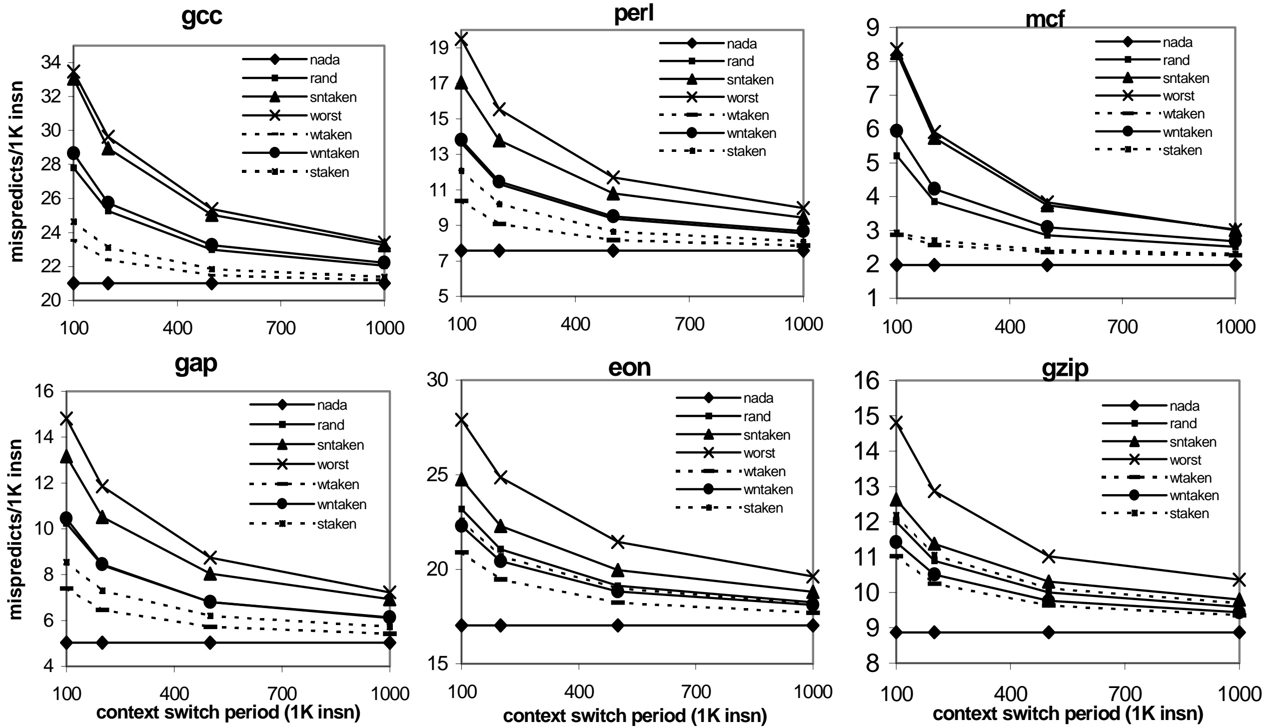

Figure 3: Mispredictions/1K instructions for various initialization schemes. *nada*: no context switches; *rand*: randomize predictor; *sntaken*: set to strongly not taken; *wntaken*: set to weakly not taken; *wtaken*: set to weakly taken; *staken*: set to strongly taken; *worst*: worst case initialization.

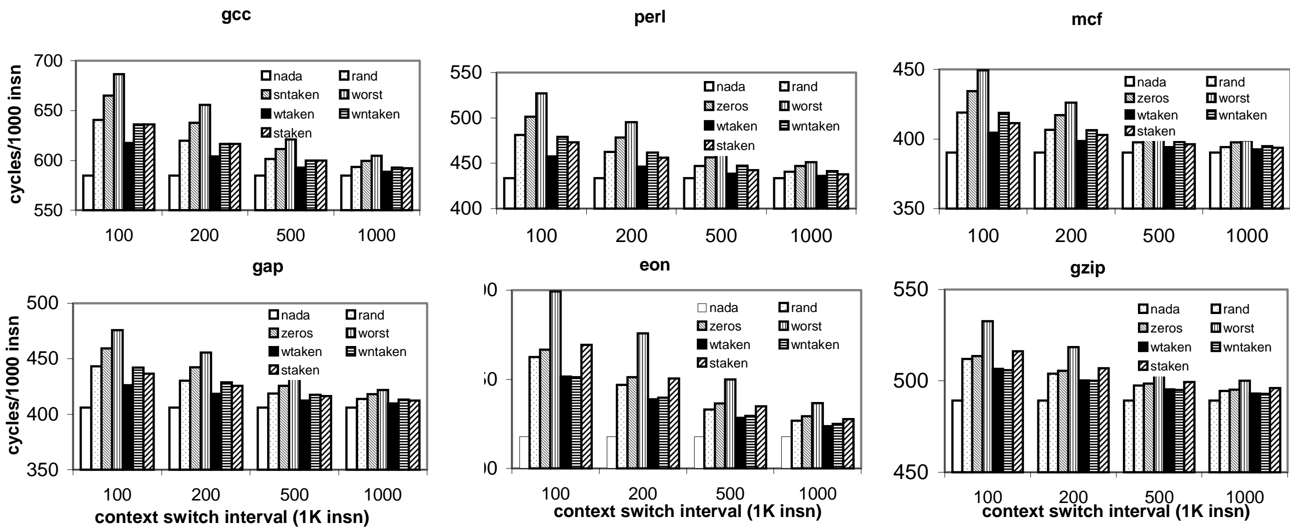

Figure 4: Performance in terms of Cycles/1K instructions for various initialization strategies.

When the predictor entries are all set to the same fixed values at a context switch, the performance varies considerably. If all the entries are initialized to strongly not taken, performance is worse than the

random case and actually approaches the "worst-case" performance for some benchmarks. In typical programs more of the branches are taken, and hence by setting all predictions to strongly not taken, we get very high performance degradation. On the other hand, setting all the predictor entries to strongly taken gives better performance, but there still are cases like Eon and Gzip, where this strategy results in worse performance than the random case. Both these strategies initialize the predictor to strongly biased values, causing two mispredictions for the branches that are predominantly biased the other way. This would suggest that initialization to weak states will lead to a significant improvement in performance. The results in Figs. 3 and 4 confirm this intuition. Initializing to weakly not taken gives a significant performance advantage compared with the random case and the strongly biased cases. With initialization to weakly taken at context switches, the number of mispredictions is cut roughly in half, i.e. to 150 to 300 total additional mispredictions per context switch. The maximum performance loss goes down to about 6.5% for 100K instruction context switch intervals and to 1.1% for 1000K instruction intervals. Hence, this performance loss will serve as a baseline upon which a VMM-based method for saving and restoring branch predictor state will improve.

4 VMM Approach

With a co-designed virtual machine, the entire predictor state can be saved and restored, so that context switches do not affect the number of mispredictions at all. However, the saving and restoring of predictor contents can lead to performance losses. For this study, we assume that all performance losses are due to code execution by the VMM, and leave other overhead effects, (e.g. pollution of the caches by the VMM) for future study, although we touch on the cache issue in the conclusions.

4.1 Simple Method

We first consider the most straightforward implementation where the entire content of the branch predictor is saved and restored. The time overhead for this save/restore is an important performance consideration. We have written assembly code and performed timing estimates for the predictor save/restore functions. We assume that one instruction can read or write 64 bits of predictor contents. The main code subsequences are 1) about 35 instructions to do initial setup and save/restore architected register values that will be used by the VMM; 2) 6 instructions to setup the save/restore loop 3) (n/64)*2 instructions to read the predictor and write the contents to memory (for a save; similar counts for a restore); n is the number of bits in the predictor. 3) (n/(64*8))*5 loop bookkeeping instructions assuming the save/restore code is unrolled 8 times. For the 8Kbit *gshare* predictor we have been assuming, this is 35+6+256+80=377 instructions. The inner loop can proceed at 2 instructions per cycle, but if we assume 1.5 instructions per cycle overall, this is about 250 cycles per save or restore. Consequently, a save/restore pair is 500 cycles. Compare this cost

with the estimated savings of 1000 to 3000 cycles per context switch; i.e. 150 to 300 mispredictions per context switch are eliminated and the misprediction penalty for each is about 9 cycles.

In order to more accurately determine performance benefits, we compare the VMM save/restore method with the best alternative option we have identified: initializing the predictor state to "weakly taken". We have (perhaps optimistically) assumed that this initialization can be done in a single cycle.

As mentioned earlier, the save/restore scheme is essentially equivalent to the case where there are no context switches. However, the VMM does have an overhead of about 500 cycles, which must be accounted for. This “adjusted performance” for the complete save/restore scheme is shown as *simple* (Fig. 5). As evident from the figure, this scheme gives performance improvements as high as 5% for 100K instruction intervals and 1% for 1000K instruction intervals, when compared with the baseline of initialize to weakly taken. The average numbers over the same intervals are 3.7% and 0.7% respectively. This indicates that a lot more cycles can be gained by eliminating context-switch-induced mispredictions than in saving and restoring the predictor contents with the VMM. In the next section we try to reduce the VMM overhead further with simple compression.

4.2 Compression Method

As just estimated, the save/restore overhead in the VMM approach is the roughly 500 cycles. One way of reducing this overhead is to compress the branch predictor contents (in a simple manner). One such approach that gives an immediate 2:1 compression is to save and restore only the direction of the prediction (and the strong/weak information). When restoring the predictor values, all entries are made “weak”. This reduces the VMM overhead to about 420 instructions or ~280 cycles. It also leads to slight branch predictor performance degradation, however.

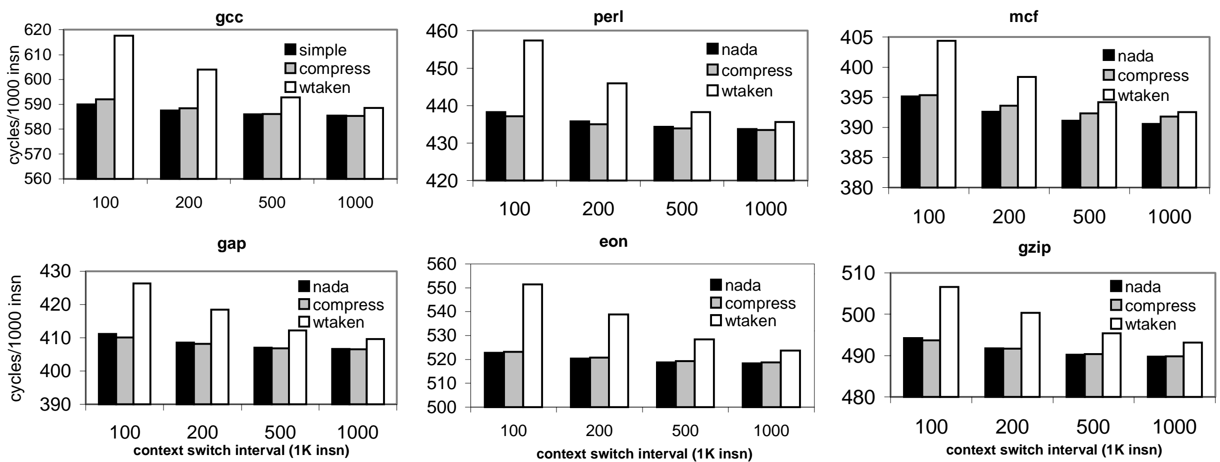

Figure 5: The bars show the performance in terms of cycles/1000 instructions for three cases - *simple*: no compression (VMM overhead 500 cycles); *compress*: 2:1 compression (VMM overhead 250 cycles); *wtaken*: weakly taken initialization (no VMM overhead)

Overall, there is a relatively small difference between the compressed and uncompressed methods (Fig. 5). The loss in performance due to increased miss prediction more or less balances the lower overhead of the VMM. The conclusion is that other engineering considerations and second order effects may determine whether compression is preferable; for example by considering the amount of hidden memory required for saving the contexts or the complexity of the branch predictor. With larger branch predictors, the balance may shift toward the compressed method.

5 Conclusions

Co-designed virtual machines provide a means for reducing overall hardware complexity by moving some of the implementation complexity into software. The VMM is in effect a micro-operating system that manages aspects of the hardware implementation. An interesting side effect is that the VMM can also be used for saving and restoring implementation state to reduce or eliminate learning times following context switches. In this paper we use the VMM to save/restore branch predictor state on context switches. When taking VMM overhead into account, there is significant performance benefit. The VMM approach leads to a 4% improvement in performance (for 100K instruction context switch intervals) when compared with initializing all entries to weakly taken.

In this study, we did not flush the caches and BTB on context switches; a preliminary study (not given here) showed that the relative performance improvement remains the same when the caches and BTB are flushed, mainly because the predictor takes longer to train compared with the BTB, and the caches have a lot of spatial locality. In any case, we did not pursue this further because we feel a better approach is to "warm up" the BTB and caches by using the VMM. In future work we plan to have the VMM record and save a number of recently used BTB entries and cache lines, and initialize/prefetch these when a context is restored.

Acknowledgements

We would like to thank Timothy Heil and Sebastian Nussbaum for several valuable discussions. This work is being supported by an SRC grant, NSF grant CCR-9900610, Intel and IBM. .

References

[1] Doug Joseph and Dirk Grunwald, "Prefetching using Markov Predictors," *Proc. of the 24th Intl. Sym. on Computer Architecture*, pp. 252-263, June 1997.

[2] R. Balasubramonian, David Albonesi, Alper Buyuktosunoglu and Sandhya Dwarkadas, "Memory Hierarchy Reconfiguration for Energy and Performance in General Purpose Architectures," *Proc. of* 33rd *Intl. Sym. on Microarchitecture*, pp. 245-257, Dec. 2000.

[3] R. Iris Bahar and Srilatha Manne, "Power and Energy Reduction via Pipeline Balancing," *Proc. of the 28th Intl. Sym. on Computer Architecture*, July 2001.

[4] Daniele Folegnani and Antonio González, "Reducing Power Consumption of the Issue Logic," *Proc. of the Workshop on Complexity-Effective Design held in conjunction with ISCA 2000,* June 10, 2000.

[5] Alper Buyuktosunoglu, David Albonesi, Stanley Schuster, David Brooks, Pradip Bose and Peter Cook, "A Circuit Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors," *Proc. of the on Great lakes Sym. on VLSI*, pp. 73-78, 2001.

[6] A. Klaiber, "The Technology Behind Crusoe Processors," Transmeta Technical Brief, http://www.transmeta.com/dev, Jan. 2000.

[7] K. Ebcioglu and E. R. Altman, "DAISY: Dynamic Compilation for 100% Architecture Compatibility, *Proc. of the 24th Intl. Sym. on Computer Architecture*, pp. 26-37, June 1997.

[8] C. F. Webb and J. S. Liptay, "A High-Frequency Custom CMOS S/390 Microprocessor," *IBM Journal of Research. and Development*, July 1997.

[9] Yuan Chou and John Paul Shen, "Instruction Path Coprocessors," *Proc. of the 27th Intl. Sym. on Computer architecture*, pp. 270-281, 2000.

[10] T. M. Conte, B. A. Patel, and J. S. Cox, "Using Branch Handling Hardware to Support Profile-Driven Optimization," *Proc. of the 27th Intl. Sym. on Microarchitecture*, pp. 12-21, Nov. 1994.

[11] Ravi Nair, "Dynamic Path-Based Branch Correlation", *Proc. of the 28th Intl. Sym. on Microarchitecture*, pp. 15-23, 1995.

[12] Marius Evers, Po-Yung Chang and Yale N. Patt, "Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches," *Proc. of the 23rd Intl. Sym. on Computer Architecture*, pp. 3-11, 1996.

[13] Toni Juan, Sanji Sanjeevan and Juan J. Navarro, "Dynamic history-length fitting: a third level of adaptivity for branch prediction," *Proc. of the 25th Intl. Sym. on Computer Architecture*, pp. 155 – 166, 1998.

[14] S. McFarling, "Combining Branch Predictors," Technical Note TN-36, Western Research Laboratory, 1993.

[15] D. Burger and T. M. Austin "The SimpleScalar Tool Set, Version 2.0," *University of Wisconsin-Madison Computer Sciences Department Technical Report #1342*, June 1997.