Dynamic Binary Translation for Accumulator-Oriented Architectures

Ho-Seop Kim and James E. Smith
Department of Electrical and Computer Engineering
University of Wisconsin—Madison
{hskim,jes}@ece.wisc.edu

Abstract

A dynamic binary translation system for a co-designed virtual machine is described and evaluated. The underlying hardware directly executes an accumulator-oriented instruction set that exposes instruction dependence chains (strands) to a distributed microarchitecture containing a simple instruction pipeline. To support conventional program binaries, a source instruction set (Alpha in our study) is dynamically translated to the target accumulator instruction set. The binary translator identifies chains of inter-instruction dependences and assigns them to dependence-carrying accumulators. Because the underlying superscalar microarchitecture is capable of dynamic instruction scheduling, the binary translation system does not perform aggressive optimizations or re-schedule code; this significantly reduces binary translation overhead.

Detailed timing simulation of the dynamically translated code running on an accumulator-based distributed microarchitecture shows the overall system is capable of achieving similar performance to an ideal out-of-order superscalar processor, ignoring the significant clock frequency advantages that the accumulator-based hardware is likely to have. As part of the study, we evaluate an instruction set modification that simplifies precise trap implementation. This approach significantly reduces the number of instructions required for register state copying, thereby improving performance. We also observe that translation chaining methods can have substantial impact on the performance, and we evaluate a number of chaining methods.

1. Introduction

A promising paradigm for processor development is the co-design of an instruction set architecture (ISA), a microarchitecture, and a dynamic binary translation system that cooperatively support an existing (virtual) ISA. Our research [28,41,42] is targeted at one such *co-designed virtual machine* (VM) that provides high performance by using a simple, distributed superscalar processor that tolerates increasing on-chip wire delays and is amenable to a very high clock frequency. A key element of co-designed VMs is dynamic binary translation (DBT) from the outwardly visible virtual ISA (V-ISA) to the implementation ISA (I-ISA). This paper studies several aspects of our evolving co-designed VM, with emphasis on the dynamic binary translation system. In our research, we use the Alpha instruction set as the V-ISA, and an accumulator-oriented instruction set [28] as the I-ISA. Fig. 1 illustrates the overall co-designed VM we are studying. The following two subsections provide an overview.

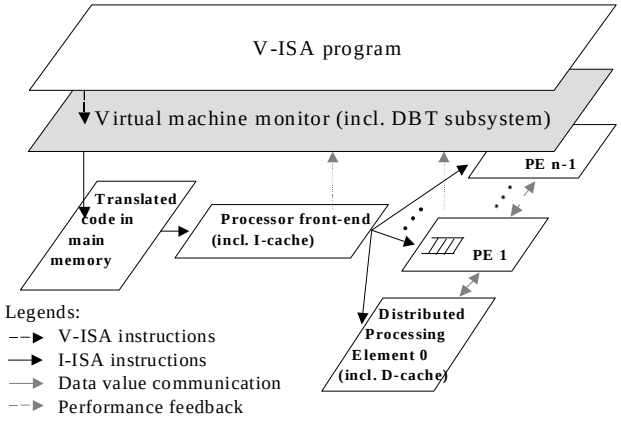

Figure 1. Dynamic binary translation system running on ILDP processor

1.1 Instruction level distributed processing

A microarchitecture trend is toward distributed, modular designs, containing partitioned issue buffers and clusters of functional units [16,25,29,37,44]. These distributed microarchitectures are designed to be tolerant of intra-processor wiring delays and contain small, fast logic subsystems. Our research is focused on ISAs and supporting implementations that naturally fit the distributed design style; i.e., *instruction-level distributed processing* (ILDP) [28,41].

The ISA proposed in [28] is based on hierarchical register architecture with a small number of accumulators on top of a larger general-purpose register (GPR) file. The use of accumulators naturally partitions the instruction stream into chains of dependent instructions, *strands*, where intra-strand register values are passed through the accumulators. Inter-strand communication is through the GPRs.

The microarchitecture consists of a pipelined instruction front-end that feeds multiple, distributed processing elements (PEs). Only the GPRs are renamed to physical registers in a conventional sense. Based on the dependence information expressed in accumulator numbers, instructions are steered to the PEs – a simple form of accumulator renaming. Each PE contains an instruction issue FIFO, a local physical accumulator, and a local copy of the GPRs. Communication among PEs is assumed to take a small number of clock cycles and is accounted for explicitly. Collectively, the hierarchical value communication, parallel in-order issue units, and distributed implementation provide overall simplicity – likely to yield a very high clock speed with moderate pipeline depth while supporting a complexity-effective form of superscalar out-of-order issue. A reorder buffer commits completed instructions in order. For more details on the microarchitecture, refer to [28].

1.2 Dynamic binary translation

Dynamic binary translation (DBT) converts instructions from a *source* ISA to a *target* ISA. In the co-designed VM paradigm [1,14,17,24,31], these are the V-ISA and I-ISA, and only the V-ISA is an existing instruction set for which conventional software exists. DBT can be performed either by a host processor [1,2,14,21, 24,26,31,34,46] or specialized hardware [10,15,18,25,40]. A DBT system can also profile program run-time behavior and dynamically optimize blocks of frequently executed instructions [3,4,6,8,11,12,15,19,30,35,36,45]. A key consideration in the design of a DBT system is the overhead resulting from translation time; any time spent translating is time not spent executing the source program.

In our DBT design the focus is on simplicity; because the underlying hardware is dynamic superscalar, it can be relied upon to provide code scheduling. The only optimization we provide is "code straightening" where basic blocks are statically located in accordance with their most common dynamic execution order (with branch directions changed accordingly). This well-known optimization, similar to static code layout techniques [38,39], captures much of the low-hanging fruit of dynamic optimization by improving instruction cache locality and branch prediction behavior.

1.3 Related work

Co-designed virtual machines were studied in the IBM DAISY [14] and BOA [1,17] projects and are implemented in the Transmeta Crusoe [24,31], all of which targeted VLIW implementations. Our research is targeted at dynamic superscalar implementations. We believe that this approach better balances the strengths of hardware and software, and results in lower overhead binary translation. Rather than trying to maximize instruction-level parallelism on a static VLIW microarchitecture using aggressive optimization and scheduling techniques, our DBT system simply identifies inter-instruction dependences and encodes the dependence information as accumulator assignments without changing the original program order. Maintaining the original instruction order greatly simplifies precise trap recovery.

Other work in this area has been targeted at dynamic binary translation from one existing instruction set to another, with code portability being the primary goal [21,26,46]. Typically code optimizations are implemented, many of them ISA-specific, and the performance "goal" is one of reducing losses; i.e. to come reasonably close to native ISA execution. Furthermore most of this work is focused on ABI translation rather than full ISA translation, as is the case with co-designed VMs.

Dynamic optimization (without translation) has also been studied [3,4,6,8,11,12,15,19,30,35,36,45]. The primary objective in this work is performance improvement, and therefore some of the techniques used, e.g. code straightening and software code caching, are related to our DBT work.

2. Accumulator-oriented I-ISAs

2.1 Basic ISA

The simplest way to describe the accumulator-oriented I-ISAs is through an example. Fig. 2 shows an Alpha code sequence translated into two different accumulator ISAs. Here we denote the accumulators as `Ai` and the GPRs as `Rj`. The modified ISA is introduced in Section 2.3. In both ISAs, the accumulators link together chains of dependent instructions. The last conditional branch instruction is translated to a combination of a conditional branch and an unconditional branch for code chaining reason; chaining is explained in Section 3.2 in more detail.

One feature of the basic ISA, introduced in [28], is that each instruction only contains one GPR, either as a source or a destination register. In this form, many instructions can be encoded using only 16 bits, reducing the code footprint. Also, addressing modes perform no address computation; this must be done with separate instructions.

2.2 Precise trap recovery in accumulator I-ISAs

The trap recovery mechanism is a fundamental aspect of any co-designed VM because it must provide exactly the same trap behavior as the V-ISA semantics define. The initial study in [28] did not consider implementation of precise traps during DBT. One goal of the research presented here is to consider the performance impact of implementing precise traps and to evaluate an I-ISA modified to simplify precise trap implementation.

There are two major issues in precise trap recovery. First, the address of the V-ISA instruction that generates trap must be identified. In our DBT system, the architected program counter is not used for actually executing the source binary code; rather an implementation program counter sequences through translated code. The

(a) Alpha assembly code

```
L1:ldbu r3, 0[r16]
   subl r17, 1, r17

   lda r16, 1[r16]

   xor r1, r3, r3
   srl r1, 8, r1
   and r3, 0xff, r3
   s8addq r3, r0,r3
   ldq r3, 0[r3]

   xor r3, r1, r1

   bne r17, L1
L2:
```

(b) Equivalent RTL notation

```
L1:R3  <- mem[R16]
   R17 <- R17 – 1

   R16 <- R16 + 1

   R3  <- R1 xor R3
   R1  <- R1 << 8
   R3  <- R3 and 0xff
   R3  <- 8*R3 + R0
   R3  <- mem[R3]

   R1  <- R3 xor R1

   P <- L1,if(R17 != 0)
L2:
```

(c) Basic I-ISA code

```
L1: A0  <- mem[R16]
    A1  <- R17 – 1
    R17 <- A1
    A2  <- R16 + 1
    R16 <- A2
    A0  <- A0 xor R1
    A3  <- R1 << 8
    A0  <- A0 and 0xff
    A0  <- 8*A0 + R0
    A0  <- mem[A0]
    R3  <- A0
    A3  <- R3 xor A3
    R1  <- A3
    P <- L1,if(A1 != 0)
    P <- L2
```

(d) Modified I-ISA code

```
L1:R3 (A0) <- mem[R16]
   R17(A1) <- R17 – 1

   R16(A2) <- R16 + 1

   R3 (A0) <- A0 xor R3
   R1 (A3) <- R1 << 8
   R3 (A0) <- A0 and 0xff
   R3 (A0) <- 8*A0 + R0
   R3 (A0) <- mem[A0]

   R1 (A3) <- R3 xor A3

   P <- L1, if (A1 != 0)
   P <- L2
```

Figure 2. Example program segment from SPEC CPU 2000 benchmark 164.gzip

address of the trapping instruction is found by indexing an address table of potentially excepting instructions (PEIs) and conditional branch instructions associated with a translation group. The address of the first V-ISA instruction in a translation group is embedded in a special instruction. This I-ISA instruction is always the first instruction in the translation group and writes the embedded address into a special register. This address provides a base for the PEI table lookup. A similar mechanism is explained in [36] in greater detail.

Secondly, all architected states must be restored to the point of the trap. Our DBT system does not reschedule instruction order; hence values are produced in the same order as the original program. However, with an accumulator ISA this is not sufficient; some source GPR values are held in an accumulator and may be overwritten prior to a trap.

One solution is to add *copy-to-GPR* instructions before instructions that overwrite an accumulator holding a value that will be live at a potential trap location. This is a fairly expensive solution, however, in terms of added instructions.

2.3 Modified ISA

An alternative to adding explicit copy instructions as just suggested is to embed GPR updates into the instruction set. The modified ISA instructions need more bits to designate the result GPRs, so some 16-bit instructions are now 32-bits. Hence, some of the "small footprint" benefit of having many 16-bit instructions [28] is lost. However, no *copy-to-GPR* instructions are necessary and most of the other advantages of the accumulator instruction set remain.

With the modified ISA, every instruction producing a result specifies a destination GPR to maintain architected state. This also means that, strictly speaking, *architected* accumulators are no longer needed. The accumulator identifiers become strand identifiers, and accumulators remain in the implementation. In effect, these strand identifiers are the dual of the independence bits in the IA-64 instruction set [23]. Nevertheless, we will typically refer to the strand identifiers as "accumulators".

To maintain implementation simplicity in the processor core, we are studying a microarchitecture where a separate register file, off the critical path, is maintained solely to keep the architected GPR state for precise traps. Only the values needed for later computation, i.e., communication, are actually written to "operational" GPRs in the critical path. Effectively, these are the same writes as in the implementation described in [28]. This is possible because the modified ISA format distinguishes two different types of register writes. Note that this scheme works in a different way from Crusoe's working/shadow registers [24,31] where all register writes go to the working register file.

In the modified instruction set example in Fig. 2d, the result registers are explicitly given and the "accumulators" (now strand identifiers) are shown in parentheses.

3. DBT for accumulator-oriented I-ISAs

In this section, we discuss DBT for the proposed accumulator-based ISAs. Subsections discuss the DBT process, including fragment formation and chaining.

3.1 Fragment formation

Our DBT mechanism follows the common two-stage (1) interpret/profile (2) translate/optimize model found in most existing dynamic optimizers and binary translators.

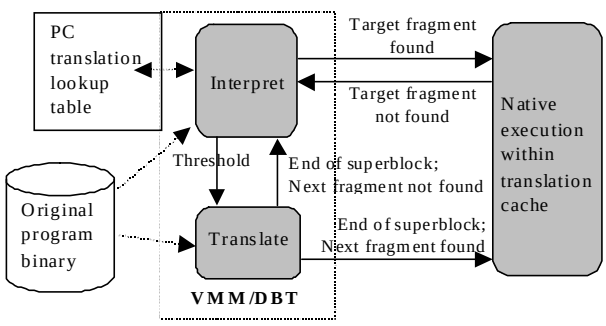

Figure 3. DBT framework

The basic unit of translation is the *superblock* [22]: a code sequence with a single entry point and multiple exit points. Following [3,4], the translated superblocks placed in a translation cache are referred to as *fragments*.

The fragment formation algorithm is a slightly modified version of Dynamo's Most Recently Executed Tail (MRET) heuristic. Our DBT system starts interpreting the given V-ISA program. Over the course of interpretation, counters are maintained for the following possible trace start candidate instructions:

- Targets of register indirect jumps (`JMP`/`JSR`/`RET` in the Alpha ISA)
- Targets of backward conditional branches
- Exit targets of existing fragments

If the number of times a static candidate instruction is executed reaches a predefined threshold, the interpreted path is followed to generate a fragment. This heuristic, a form of simple software speculation, is based on the observation that when an instruction becomes "hot", it is statistically likely that the very next sequence of executed instructions is also hot. Fragment ending conditions are:

- Register indirect jumps or trap instructions
- Backward taken conditional branches
- Already collected instructions are found again (a cycle)
- A predefined maximum number of instructions is reached

Currently our DTB algorithm does not perform loop unrolling. However, there still is code redundancy inherent in superblock-based translation.

A newly translated fragment is placed in the translation cache. If later program control flow reaches an existing fragment, the translated instructions in the fragment are executed directly on the hardware.

One important byproduct of "hot" trace-based optimizations/translations is that dynamic "code straightening" is automatically performed; i.e. basic blocks that commonly occur in sequence dynamically are placed together statically, enhancing effective fetch bandwidth.

3.2 Fragment chaining

After a translated fragment is executed, the next fragment must be found (if it exists). In a simple implementation, the fragment would branch to a "dispatch" code sequence at a certain fixed location. This dispatch code locates the next fragment starting address using the PC translation lookup table in Fig. 3. To limit the number of these translation address lookups, a good chaining mechanism that can provide next address in a fast and efficient manner is essential. Furthermore, to supplement conventional chaining, using a co-designed VM provides us the ability to implement special instructions in the I-ISA that further reduce the fragment transition overhead.

Direct branches, either conditional or unconditional, are relatively easy to handle because their (taken) target addresses do not change during program execution. Unconditional direct branches are either simply removed by code straightening (if they do not save return addresses) or are changed to special *save-V-ISA-return-address* instructions (if they save return addresses). If a conditional branch is taken at translation time, its condition is reversed to allow continuous fetching. If the superblock exit target (that is, the path not followed at translation time) is not found in the translation cache, a special *call-translator-if-condition-is-met* instruction is generated for the branch. If the target instruction becomes hot and is later translated, the DBT system replaces the *call-translator-if-condition-is-met* instruction with a normal conditional branch instruction – a "patch" is performed.

To save lookup overhead for each and every indirect jump, most dynamic optimizers/translators implement a form of software based jump target prediction. In [3,4,14,35], a sequence of instructions compares the indirect target address held in a register against an embedded translation-time target address. A match indicates a correct "prediction" and the inlined target instructions can be executed; if not, the code branches to stub code at the end of the fragment, finally reaching the shared dispatch code.

Even though software jump target prediction is often correct, it still adds overhead of additional compare-and-branch code. Worse, if the prediction fails, the dispatch code needs to be executed, anyway. Bruening et al. [6] identify this as the highest overhead and report that an indirect jump lookup takes 15 instructions, while the compare-and-branch code sequence takes 6 instructions in the x86 ISA. In our DBT, three instructions suffice for the compare-and-branch code, utilizing a special *load-embedded-target-address* instruction. The dispatch code takes 20 instructions.

Return instructions pose the most serious problem because their target addresses can change more frequently than other indirect jumps. The DAISY system [14] cuts performance losses by allowing multiple compare-and-branch codes for a single return instruction. Most modern processors contain a hardware return address stack (RAS) mechanism [27], which can predict a return instruction's target address very accurately. With dynamic trace-based DBT like ours, however, a conventional hardware return address stack cannot be utilized because it does not track the V-ISA return addresses.

We propose a specialized hardware RAS mechanism that contains pairs of V-ISA return addresses and their corresponding I-ISA return addresses. When a return instruction is fetched, the next fetch address is predicted by the popped translated return address. The V-ISA address is checked against to the instruction's register value. If they match, the prediction was successful. If they don't, control is redirected to the dispatch code by an unconditional branch following the return instruction. Note that the semantics of return instructions are slightly changed

from the conventional definition – a traditional return instruction always jumps to the target atomically. To push the return address pair, a special *push-dual-address-RAS* instruction is generated for a V-ISA instruction (BSR and JSR in Alpha ISA) to save return addresses.

3.3 DBT algorithm

The major function of the translation process is identifying strands and re-mapping intra-strand communication values to accumulators. Because of the nature of dynamically constructed traces, there is no need for graph-traversing dependence analysis usually found in static compilers. Below is the high-level outline of the translation algorithm. Although the algorithm appears to contain multiple passes of sequential scans, they can be combined to a single pass.

Dependence/usage identification: when a hot code sequence is found by interpretation, an instruction's dependences on previous instructions in the same superblock are identified. In addition, the usage of input and output registers is examined to determine the "globalness" of their values. Global values are placed in the globally visible GPRs. Following are the important usage categories.

- **No user**: an output register value is not used before being overwritten. An instruction whose output value is not used naturally ends a strand.
- **Local**: an output register value used only once before being overwritten in the same superblock. These are candidates for assignment to accumulators.
- **Temp**: values passed between two decomposed instructions (e.g., conditional moves). These are assigned to accumulators.
- **Live-in global**: input register values that are live on superblock entry; assigned to GPRs.
- **Live-out global**: output register values live on superblock exit; assigned to GPRs.
- **Communication global**: register values used more than once before being overwritten in the same superblock; assigned to GPRs.
- **Spill global**: a) If an instruction has two local input registers, one is made a spill global because the I-ISA does not allow two different accumulators in the same instruction. b) If a strand has to be terminated to free an accumulator (see below), a local value is converted to a spill global.

Strand formation: based on dependence and input register usage patterns, instructions are scanned and a strand number is assigned to each instruction. The translator uses an unlimited number of strands that are later assigned to a finite number of accumulators. Here temp usage is treated the same way as local. If an instruction has:

- Zero local input registers: a strand is started and a new strand number is assigned to it. Furthermore, instructions with two global input registers are broken into two accumulator instructions – a *copy-from-GPR* instruction and a translated source instruction that uses the (local) result of the copy as an input. The copy instruction initiates a strand, and the source instruction now has one local input register (and is handled accordingly).
- One local input register: assigned the same strand number as the instruction producing the local value.
- Two local input registers: a heuristic is needed to decide which strand number to assign. If one of the input registers is a temp, then the temp producer's strand number is assigned. Otherwise the number assigned corresponds to the longer strand up to that point (length is determined by instruction count).

Accumulator assignment: The strand numbers are converted to finite accumulator numbers. Instead of using a traditional graph coloring heuristic to assign accumulator numbers to strands, a simple linear-scan heuristic is used. When the translator runs out of accumulators, a live strand is chosen for termination and the accumulator is freed. This is done by inserting a *copy-to-GPR* instruction at the strand termination point, and a *copy-from-GPR* instruction before the resumption point.

4. Evaluation

4.1 Methodology

Our aim is to evaluate performance impact of the proposed I-ISAs on the ILDP microarchitecture. In a real co-designed VM, the DBT itself (and all the other VM software) would execute the I-ISA. In our current simulation-based environment this is impractical. Consequently, the DBT system is written in C and is compiled to execute on the simulation platform (SimpleScalar 3.0C [7] running on Intel clusters). Nevertheless, the simulated DBT system does perform all interpretation and binary translation in the same sequence as it would in a co-designed VM.

Another simulation constraint is that the benchmarks we run have relatively short executions times compared with real applications. Consequently, the interpretation and translation overheads, although small, are still disproportionately large for some of the benchmarks. In other systems [3,14], interpretation and translation/ optimizations overhead have been found to be very small.

With respect to interpretation, we do nothing special. The same techniques as used by others [3,5,14,32,33] will suffice, and the interpretation overhead should be about

the same. If an instruction is interpreted 50 times (the threshold value) and each interpretation takes about 20 instructions, this is a total of about 1,000 target instructions per source instruction.

Typically, binary translation overhead is an order of magnitude higher than interpretation; that is, thousands of instructions per translated source instruction [14]. As pointed out earlier, the translation overhead for our proposed system should be relatively low, and we evaluate this overhead in Section 4.2.

To evaluate performance, we focus on detailed simulation of all *translated* code, including all chaining code. Our simulation system does switch between interpretation, translation, and execution modes, however. When the simulated program control flow reaches a previously translated fragment, the simulator begins detailed timing simulation. Here the timing simulation starts with an initially empty pipeline. Similarly if an exit condition from the timing simulator is met (i.e., a target fragment was not found in translation cache, indicated by a *call-translator* instruction), the mode is changed to interpretation after the last instruction in the pipeline is committed. Overall performance is then measured as V-ISA instructions per cycle (IPC) for execution of all translated (and chained) instructions.

To evaluate performance we use three DBT/simulators as well as a slightly modified superscalar simulator (referred to as "original" in later sections). Two of these perform translation from Alpha to basic and modified accumulator ISAs, respectively. The third converts an Alpha binary to a code-straightened version of Alpha and simulates the same superscalar microarchitecture as the "original"; this is for isolating the effects of code straightening and fragment chaining from other DBT performance effects. This third DBT/simulator tool is referred to as the "code-straightening-only" simulator in later subsections. Table 1 lists microarchitecture parameters of two hardware platforms simulated. Modifications to the superscalar processor simulator are marked with *.

To collect statistics, we use the SPEC CPU2000 integer benchmarks [20] compiled for Alpha EV6 ISA at the base optimization level (`–arch ev6 –non_shared –fast`). The `–fast` option includes `–O3` optimization level, even for base optimization. The compiler flags are same as those reported for Compaq AlphaServer ES40 SPEC CPU2000 benchmark results. DEC C++ V.6.1-027 (for *252.eon*) and C V.5.9-005 (for the rest) compilers were used on Digital UNIX 4.0-1229. The `test` input set was used for all benchmarks except for *253.perlbmk*, where one of the `train` input set (`-I./lib diffmail.pl 2 550 15 24 23 100`) was used. All benchmarks were run up to completion or 4.3 billion instructions. Skipping the initialization phase and simulating only part of program using `ref` input sets, as is frequently done in microarchitecture research, would likely have exaggerated the benefits of DBT.

For translation, we use a maximum superblock size of 200 and a threshold of 50. We also experimented with superblock size of 50 and found it is not large enough to provide performance benefits from code straightening. We use only four logical accumulators (as opposed to eight used in [28]). We found that four accumulators are generally sufficient, and few strands must be prematurely terminated to free up an accumulator. Four logical accumulators are used throughout the evaluation except where noted.

In this study, we use an unlimited number of counters for superblock start candidates. For programs the size of the SPEC benchmarks, however, the number of counters is relatively small. Also, the static code size of the SPEC benchmarks (all are less than 1Mbytes except *176.gcc* at 2.3Mbytes) means that they would comfortably fit into a reasonably sized translation cache. Thus translation cache management is not required; however, other research indicates that this overhead is generally negligible [4]. In fact, there may be a performance *cost* in not occasionally flushing translation cache entries. Some fragments may be sub-optimal because once a fragment is constructed

Table 1. Microarchitecture parameters

	Out-of-order superscalar (for original and code-straightening-only simulators)	ILDP microarchitecture (basic and modified accumulator ISAs)
Branch prediction	16K entry, 12-bit global history g-share predictor, 8-entry RAS, 512-entry, 4-way set-assoc. BTB 3-cycle fetch redirection latencies for both misfetch and misprediction	
I-cache	128-byte line size, direct-mapped, 32-KB size, LRU replacement; up to 3 sequential basic blocks*	
D-cache	64-byte line size, 4-way set-assoc., 32-KB size, 2-cycle latency, random replacement	Same as left or 64-byte line size, 2-way set-assoc., 8-KB size, 2-cycle latency, random replacement; replicated across PEs
L2 cache	128-byte line size, 4-way set-assoc., 1-MB size, 8-cycle latency, random replacement	
Memory	72-cycle latency, 64-bit wide, 4-cycle burst	
Reorder buffer size	128 Alpha instructions	128 ILDP instructions
Decode/retire bandwidth	4 Alpha instructions	4 ILDP instructions
Issue window size	128 (same as the reorder buffer size)	4/6/8 (FIFO heads)
Issue bandwidth	4	4/6/8
Execution resources	4 fully symmetric functional units	4/6/8 fully symmetric functional units
Misc.	No communication latency, oldest-first issue*	2 or 0 cycle global communication latency

there is no second chance for forming different fragment starting from the same address in our current implementation. In Dynamo, for example, the fragment cache is flushed when there is change of program phase (indicated by abrupt increase of fragment generation rate) thereby evicting infrequent fragments from the cache and allowing new fragment formation.

4.2 Translation overhead

To measure the translation overhead, we compiled the DBT/simulator tool onto the Alpha ISA and built an instrumented version with Atom tools on an Alpha 21164 system running Tru64 5.1A. The instrumented version was then run with the same SPEC benchmarks. The number of dynamic instructions (in Alpha) is reported for each procedure. Averaging the total number of instructions executed in translation-related procedures with the number of translated instructions gives the average number of DBT instructions required to translate a single instruction.

On average, 1,125 Alpha instructions were executed to translate a single Alpha instruction. Table 1 in Section 4.4 contains per-benchmark results. This number is about a quarter of the 4,000+ PowerPC instructions needed to translate a PowerPC instruction into VLIW architecture, as reported in [14]. Of course, most of this reduction comes from the simple nature of our DBT – no aggressive optimization is performed.

Our DBT code in its current form was written for flexibility of simulation, not for translation speed. For example, on average, 20% of the instructions are spent copying high-level data structures that represent translated instructions to the translation cache structure, field by field. We feel that with optimizations for translation speed we can reduce the number of instructions per translated instruction to well below 1,000.

4.3 Evaluation of fragment chaining methods

We used the code-straightening-only simulator to evaluate the effect of fragment chaining. Chained fragments show different branch/jump prediction behavior from the original source binary. For example, a single register-indirect jump in the source is now replaced with a compare-and-branch sequence plus optional dispatch code (contains another jump). That is, the number and type of control-transfer instructions are changed by fragment chaining. To estimate the performance effect of different fragment chaining options, we tested three different implementations against the "original" simulator. In Fig. 4, predictor performance is measured as the number of branch/jump mispredictions per 1,000 instructions.

In the first implementation *no_pred*, software prediction is not used; a branch to a shared dispatch code is always generated for a register indirect jump. The dispatch code performs a PC translation table lookup to find the next fragment. Target address prediction rate of the indirect jump at the end of the dispatch code is expected to be very poor because all indirect jumps lead to the same dispatch code and a single BTB entry is required to provide all the target address predictions.

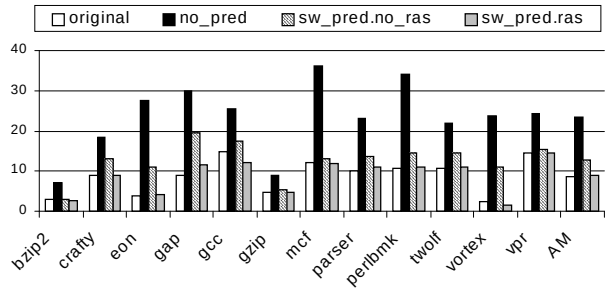

Figure 4. Number of mispredictions per 1,000 instructions

The second implementation, *sw_pred.no_ras*, uses conventional software prediction as found in [3,4,14,35]. This reduces mispredictions substantially by reducing the probability of executing the shared dispatch code. Although the number of mispredictions was reduced to almost half of the *no_pred*, it is still 45% more than the original Alpha code. This shows the inherent limit of the simple translation-time prediction.

The final implementation, *sw_red.ras*, uses a specialized dual-address hardware RAS described at the end of Section 3.2. This method achieves nearly the same number of mispredictions as the original Alpha code. This implementation is used as baseline in later simulations.

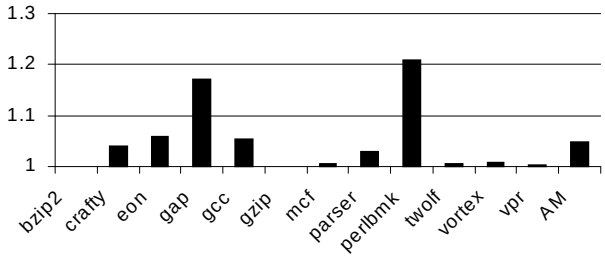

Figure 5. Relative instruction count

Fig. 5 shows instruction count expansion from chaining. Even with efforts to reduce dynamic instruction count increased by indirect jump chaining such as software prediction and dual-address RAS (returns are not translated into compare-and-branch sequence), several benchmarks still experience serious expansion. In the benchmarks where the indirect jump code expansion rate is small, procedure calls are mostly performed by unconditional direct branches (BSR in Alpha ISA).

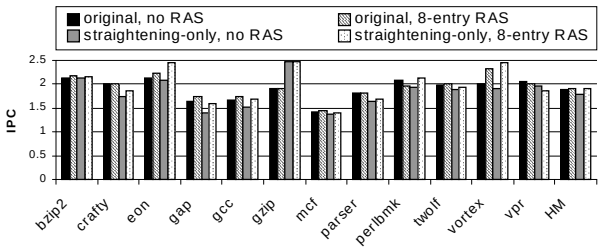

Figure 6. Performance impact of code straightening and H/W RAS

In Fig. 6, code-straightening-only performance is compared to original program performance. The straightened version without RAS performs *worse* compared to the original program without RAS. Here the benefit of code straightening is offset by more branch address mispredictions and an increased number of instructions executed in the compare-and-branch and dispatch code. On the other hand, our baseline model with dual-address RAS performs about the same level as the original program with RAS. This is an example of special hardware features that can be specifically designed for co-designed system. Nevertheless, the performance improvement by code straightening is less than [4] suggests. One important factor is that a dynamic optimizer like Dynamo can choose to "bail out" if performance improvement is not found while a DBT system cannot. If we exclude underperforming 6 benchmarks (where a bail out would be used), 7% IPC *improvement* is observed.

4.4 DBT characteristics

We now consider characteristics of translated code. Since NOP instructions are removed by translation, they are not counted in V-ISA (Alpha) program characteristics. In Table 2, B and M denote basic ISA and modified ISA respectively.

Table 2. Translated instruction statistics

	Relative number of dynamic instructions		% of copy instructions		Relative number of static instruction bytes		No. of Alpha inst to translate a Alpha inst
	B	M	B	M	B	M	
bzip2	1.58	1.30	18.9	1.3	1.07	1.02	818.3
crafty	1.59	1.38	17.4	5.0	1.13	1.08	1147.0
eon	1.77	1.51	18.6	4.6	1.33	1.15	1259.0
gap	1.62	1.40	15.2	1.5	1.10	1.05	N/A
gcc	1.56	1.34	15.0	1.1	1.13	1.09	1133.0
gzip	1.49	1.23	21.7	4.8	1.13	1.06	989.3
mcf	1.62	1.35	19.8	3.7	1.16	1.10	N/A
parser	1.55	1.26	19.8	1.4	1.11	1.07	966.7
perl-bmk	1.74	1.54	12.2	1.1	1.11	1.08	1829.0
twolf	1.56	1.29	20.9	4.9	1.16	1.06	1235.1
vortex	1.47	1.33	11.8	2.1	1.11	1.08	658.2
vpr	1.59	1.33	20.9	5.6	1.14	1.04	1211.6
Avg.	**1.60**	**1.36**	**17.7**	**3.1**	**1.17**	**1.07**	**1124.7**

The second and third columns show the increase in dynamic instructions compared with Alpha code. The basic ISA shows of increase of 60% on average, while the modified ISA has only about half the increase (36%). The modified ISA has significantly fewer translated instructions because it does not generate *copy-to-GPR* instructions to maintain V-ISA architected state. This is shown in columns 4 and 5, where only 3.1% of the instructions are copy instructions in the modified ISA compared with 17.7 percent for the basic ISA. The rest of the instruction count expansion comes mostly from chained register-indirect jump and decomposed memory instructions.

On the other hand, both I-ISAs are relatively good in code size expansion rate. The use of 16-bit instructions and special branch instructions helps here. The code expansion rates are 17% for the basic ISA and 7% for the modified ISA.

The output register value type is an important program characteristic because it affects number of extra copy instructions and write ports requirement of the general purpose register file.

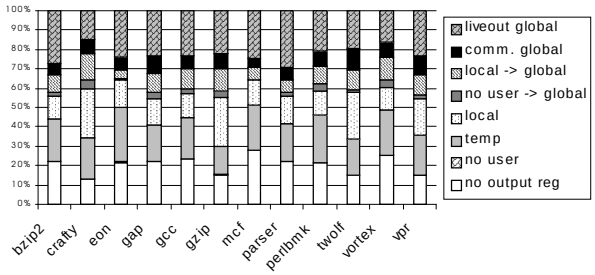

Figure 7. Output register usage

Output register value usage ("globalness") statistics in Fig. 7 was collected for Alpha instructions in superblocks. Here, memory instructions with effective address calculation are decomposed into two nodes.

For modified ISA, about 25% of dynamic instructions have a global output value (liveout and communication globals). These are the global values that should be written to latency-critical GPRs. The rest of the produced register values go to the local accumulator; these are also used to update architected state, *off* the critical path of the processor pipeline.

Using the basic ISA requires extra *copy-to-GPR* instructions to maintain state at fragment exit points including conditional branches. The notation "local → global" represents values that are used only once but need to be saved to a GPR before a conditional branch. The notation "no user → global" is similar. If these values (representing extra *copy-to-GPR* instructions in the basic ISA format) are included, the total percentage of instructions that have global output values rises to about 40%.

Those statistics are in contrast to earlier results [28] where, for an oracle program trace (e.g., register values do not need to be saved at superblock boundaries), we found only 20% of instructions produce global output values.

4.5 IPC performance

Our overall goal in the research reported here is to achieve IPC rates that are close to native Alpha performance. The eventual objective of our research is to achieve performance gains due to a simple and high clock frequency microarchitecture provided by the accumulator-oriented ISA [28].

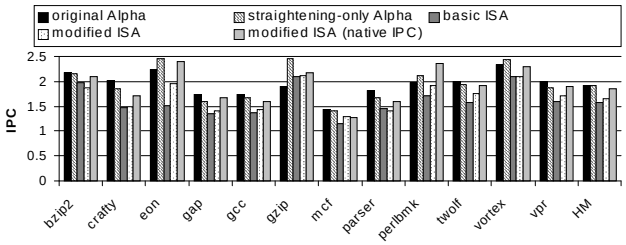

Figure 8. IPC comparison

Fig. 8 compares performance of the ILDP processor running dynamically translated accumulator ISA code (the third and fourth bars) with the conventional superscalar processor running original code (the first bar) and code-straightened version (the second bar). These four bars show V-ISA instructions per cycle (IPC). For the ILDP microarchitecture, 8-way PEs, 32KB L1 data cache (same size as the superscalar processor), and 0-cycle global communication latency were used; this is to isolate the I-ISA effects from the machine resources.

The difference in performance between the straightening-only Alpha and the modified ISA is a measure of the IPC cost in using the simple accumulator-based ISA. Many factors contribute to this IPC performance loss of 15%. The most important is about 36% more instructions; a much higher native I-ISA IPC, shown in the last bar, is clearly offset by the extra instructions. One way to deal with this instruction count expansion is to not split memory instructions into two. This puts more pressure on decoding hardware but nonetheless reduces pressure on fetch and reorder buffer mechanisms. However store instructions have two input registers and as such, sometimes have to be split. Register indirect jump chaining is another area to look for further optimization, especially with further specialized hardware support. Lastly, even certain run-time optimizations can be considered if they do not affect the simple trap recovery model.

In the ILDP microarchitecture, L1 data cache is replicated across the PEs. Although this replication allows faster L1 data cache access by reducing the number of ports per cache, both microarchitectures were given same data cache latency.

Another important point is rather idealistic execution model of the reference superscalar processor. For example, an instruction issue window size of 128 is probably not realizable with today's clock frequency requirements. A recent study [13] showed that the SimpleScalar simulator used in this study reports about 20% more IPC compared to a detailed microarchitecture simulator that models various hardware resource limits.

Therefore, the more relevant point is how well the proposed co-designed VM handles technology constraints such as increased global wire latencies. In Fig. 9, IPC variation from L1 data cache size (32-KB and 8-KB), number of processing elements (four, six, and eight), and global wire latency (zero and two) are shown.

We experimented with 8 logical accumulators to see if more than 4 logical accumulators are necessary. The first bar in Fig. 9 shows that 8 logical accumulators do help (11% IPC improvement). However, to support 8 logical accumulators, there must be at least 8 PEs to guarantee the pipeline is deadlock-free. A minimum 8-way backend seems like an excessive investment considering that the typical IPC of SPEC INT type workload is below 2. Another point is that even a single bit in the ISA format space is a very precious resource. That one bit can be useful especially for opcode fields in 16-bit instructions.

A quarter size L1 data cache (compare the second and the third bars) does not affect the overall performance very much, at least for `test` runs of SPEC CPU2000 benchmarks used in our evaluation. More importantly, only 3.4% of IPC loss is observed by adding two cycle

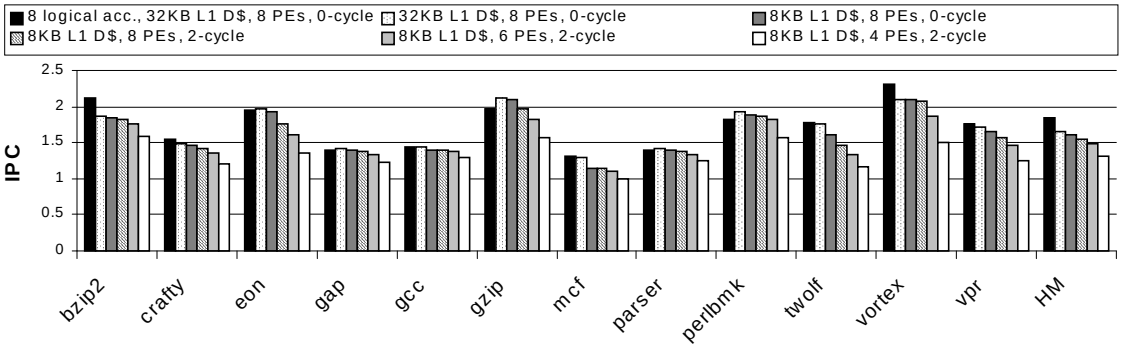

Figure 9. IPC variation over machine parameters

communication latencies (the fourth bar). This shows that dependence-based strand identification and simple steering based on accumulator numbers works well to tolerate the inter-PE communication latency. Lastly, it is observed that 6-PE configuration holds up fairly well (5% loss compared to 8-PE), while 4-PE lags 8-PE by 18%.

5. Conclusion

We have presented a dynamic binary translation system for a co-designed virtual machine. The accumulator-oriented implementation instruction set is designed to accommodate simple, high clock frequency microarchitectures that are tolerant to increasing global wire latencies. Translation overhead is dramatically reduced compared to previous DBT systems based on VLIW processors. No aggressive optimization techniques are used because the underlying hardware is capable of dynamic instruction rescheduling between simple in-order FIFOs. Rather than relying heavily on hardware to achieve high instruction-level parallelism, our DBT software simply helps underlying microarchitecture tolerate ever-increasing global wire latencies while providing the benefits of dynamic code straightening – one of the most efficient dynamic performance enhancement techniques. It is our belief that the combination of simple, distributed microarchitecture capable of dynamic instruction scheduling and low-overhead dynamic binary translation provides a good design tradeoff point.

We studied two I-ISA forms and found the modified format with destination register specifier provides both a simpler exception recovery model and higher performance compared to the basic format. We identify fragment chaining overhead as the one of the biggest performance limiting factors that offsets the benefits of code straightening. Fragment chaining overhead is exacerbated in our DBT because no other aggressive optimizations are performed to reduce the number of executed instructions. Dual address return address stack, a proposed co-designed VM feature, greatly helps reducing misprediction rate as well as instruction count expansion from chaining.

6. Acknowledgements

We would like to thank Ashutosh S. Dhodapkar for his help on the initial version of the dynamic binary translator. This work is being supported by SRC grant 2001-HJ-902, NSF grants EIA-0071924 and CCR-9900610, Intel and IBM.

7. References

[1] Erik R. Altman, Michael Gschwind, Sumedh Sathaye, S. Kosonocky, Arthur Bright, Jason Fritts, Paul Ledak, David Appenzeller, Craig Agricola, Zachary Filan, "BOA: The Architecture of a Binary Translation Processor," *IBM Research Report RC 21665*, Dec. 2000

[2] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, Peter F. Sweeney, "Adaptive Optimization in the Jalapeno JVM," *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 47-65, Oct. 2000.

[3] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, "Transparent dynamic optimization: the design and implementation of Dynamo," *Hewlett Packard Laboratories Technical Report HPL-1999-78*, Jun. 1999.

[4] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, "Dynamo: A Transparent Dynamic Optimization System," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1-12, Jun. 2000.

[5] James R. Bell, "Threaded Code," *Communications of ACM*, Vol. 16, No. 6, pp. 370-372, Jun. 1973.

[6] Derek Bruening, Evelyn Duesterwald, Saman Amarasinghe, "Design and Implementation of a Dynamic Optimization Framework for Windows," *Proceedings of the 4th Workshop on Feedback-Directed and Dynamic Optimization*, Dec. 2001.

[7] Douglas C. Burger and Todd M. Austin, "The SimpleScalar Toolset, Version 2.0," *Technical Report CS-TR-97-1342*, University of Wisconsin—Madison, Jun. 1997.

[8] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, David M. Gillies, "Mojo: A Dynamic Optimization System," *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, Dec. 2000.

[9] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, John Yates, "FX!32 - A Profile-Directed Binary Translator," *IEEE Micro*, Vol. 18, No. 2, pp. 56-64, Mar. 1998.

[10] Yuan Chou, John. P. Shen, "Instruction Path Coprocessors," *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 270-281, Jun. 2000.

[11] Thomas M. Conte, Sumedh W. Sathaye, "Dynamic Rescheduling: A Technique for Object Code Compatibility in VLIW Architectures," *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 208-218, Dec. 1995.

[12] Dean Deaver, Rick Gorton, Norman Rubin, "Wiggins/Redstone: an online program specializer," *Proceedings of the 11th HotChips Symposium*, Aug. 1999.

[13] Rajagopalan Deskian, Douglas C. Burger, Stephen W. Keckler, "Measuring Experimental Error in Microprocessor Simulation," *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 266-277, Jun 2001.

[14] Kemal Ebcioglu, Erik Altman, Michael Gschwind, Sumedh Sathaye, "Dynamic Binary Translation and Optimization," *IEEE Transactions on Computers*, Vol. 50, No. 6, pp. 529-548, Jun. 2001.

[15] Brian Fahs, Satarupa Bose, Matthew Crum, Brian Slechta, Francesco Spadini, Tony Tung, Sanjay J. Patel, Steven S. Lumetta, "Performance Characterization of a Hardware Mechanism for Dynamic Optimization," *Proceedings of the 34th International Symposium on Microarchitecture*, pp. 16-27, Dec. 2001.

[16] Keith Farkas, Paul Chow, Norman Jouppi, Zvonko Vranesic, "The Multicluster Architecture: Reducing Cycle Time Through Partitioning," *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 40-51, Dec. 1997.

[17] Michael Gschwind, Erik R. Altman, Sumedh Sathaye, Paul Ledak, David Appenzeller, "Dynamic and Transparent Bi-

nary Translation," *IEEE Computer*, Vol. 33, No. 2, pp. 54-59, Mar. 2000.

[18] Linley Gwennap, "Intel's P6 Uses Decoupled Superscalar Design," *Microprocessor Report*, Feb. 16, 1995.

[19] Kim M. Hazelwood, Thomas M. Conte, "A Lightweight Algorithm for Dynamic If-Conversion During Dynamic Optimization," *Proceedings of the 2000 International Symposium on Parallel Architectures and Compilation Techniques*, pp. 71-80, Oct. 2000.

[20] John L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *IEEE Computer*, Vol. 33, No. 7, pp. 28-35, Jul. 2000.

[21] Raymond J. Hookway, Mark A. Herdeg, "Digital FX!32: Combining Emulation and Binary Translation," *Digital Technical Journal*, Vol. 9, No. 1, Jan. 1997.

[22] Wen-mei W. Hwu, , Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *Journal of Supercomputing*, Kluwer Academic Publishing, pp. 229-248, 1993.

[23] Intel Corp., Intel Itanium Architecture Software Developer's Manual vol. 3, Rev. 2.0: Instruction Set Reference, *Intel Corp.*, 2001.

[24] Tom R. Halfhill, "Transmeta Breaks x86 Low-Power Barrier," *Microprocessor Report*, Feb. 14, 2000.

[25] Glenn Hinton, Dave Sager, Mike Upton, Darrel Boggs, Doug Carmean, Alan Kyker, Patrice Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal Q1*, 2001.

[26] Paul Hohensee, Mathew Myszewski, David Reese, "Wabi CPU Emulation," *Proceedings of the 8th HotChips Symposium*, pp. 47-65. Aug. 1996.

[27] David Kaeli, P. G. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns," *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 34-42, Jun. 1991.

[28] Ho-Seop Kim, James E. Smith, "An Instruction Set and Microarchitecture for Instruction-Level Distributed Processing," *Proceedings of the 29th International Symposium on Computer Architecture*, pp. 71-81, Jun. 2002.

[29] Richard E. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro*, Vol. 19, No. 2, pp. 24-36, Mar. 1999.

[30] Thomas Kistler, Michael Franz, "Continuous Program Optimization: Design and Evaluation," *IEEE Transactions on Computers*, Vol. 50, No. 6, pp. 549-565, Jun. 2001.

[31] Alexander Klaiber, "The Technology behind Crusoe Processors," *Transmeta Technical Brief*, 2000.

[32] Paul Klint, "Interpretation Techniques," *Software Practice and Experience*, Vol. 11, No. 9, pp. 963-973, Sep. 1981.

[33] Peter S. Magnusson, David Samuelsson, "A Compact Intermediate Format for SIMICS," *Technical Report R94:17*, Swedish Institute of Computer Science, 1994.

[34] Steve Meloan, "The Java HotSpot Performance Engine: An In-Depth Look," *Technical Whitepaper*, Sun Microsystems, 1999.

[35] Matthew Merten, Andrew R. Trick, Ronald D. Barnes, Erik M. Nystrom, Christopher N. George, John C. Gyllenhaal, Wen-mei W. Hwu, "An Architectural Framework for Run-Time Optimization," *IEEE Transactions on Computers*, Vol. 50, No. 6, pp. 567-589, Jun. 2001.

[36] Erik Nystrom, Ronald D. Barnes, Matthew C. Merten, and Wen-mei W. Hwu, "Code Reordering and Speculation Support for Dynamic Optimization Systems," *Proceedings of the Int. Conference on Parallel Architectures and Compilation Techniques*, Sep. 2001.

[37] Subbarao Palacharla, Norman P. Jouppi, James E. Smith, "Complexity-Effective Superscalar Processors," *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 206-218, Jun. 1997.

[38] Karl Pettis, Robert C. Hansen, "Profile Guided Code Positioning," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 16-27, Jun. 1990.

[39] Alex Ramire, Josep L. Larriba-Pey, Carlos Navarro, Josep Torrellas, Matero Valero, "Software Trace Cache," *Proceedings of the 13th International Conference on Supercomputing*, pp. 119-126, Jun. 1999.

[40] Michael Slater, "AMD's K5 Designed to Outrun Pentium," *Microprocessor Report*, Oct. 24, 1994.

[41] James E. Smith, "Instruction-Level Distributed Processing," *IEEE Computer*, Vol. 34, No.4, pp. 59-65, Apr 2001.

[42] James E. Smith, S. Subramaya Sastry, Timothy H. Heil, Todd M. Bezenek, "Achieving High Performance via Co-Designed Virtual Machines," *International Workshop on Innovative Architecture*, Maui High Performance Computer Center, Oct. 1998.

[43] Michael D. Smith, "Overcoming the Challenges to Feedback-Directed Optimization," *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, Dec. 2000.

[44] Joel M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le, B. Sinharoy, "POWER4 System Microarchitecture," *IBM Journal of Research and Development*, Vol. 46, No. 1, pp.5-26, Jan. 2002.

[45] David Ung, Cristina Cifuentes, "Optimizing Hot Paths in a Dynamic Binary Translator," *Proceedings of the 2nd Workshop on Binary Translation*, Oct. 2000.

[46] Cindy Zheng, Carol Thompson, "PA-RISC to IA-64: Transparent Execution, No Recompilation," *IEEE Computer*, Vol. 33, No. 3, pp. 47-53, Mar. 2000.