Hardware Support for Control Transfers in Code Caches

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

Abstract

Many dynamic optimization and/or binary translation systems hold optimized/translated superblocks in a code cache. Conventional code caching systems suffer from overheads when control is transferred from one cached superblock to another, especially via register-indirect jumps. The basic problem is that instruction addresses in the code cache are different from those in the original program binary. Therefore, performance for register-indirect jumps depends on the ability to translate efficiently from source binary PC values to code cache PC values.

We analyze several key aspects of superblock chaining and find that a conventional baseline code cache with software jump target prediction results in 14.6% IPC *loss* versus the original binary. We identify the inability to use a conventional return address stack as the most significant performance limiter in code cache systems. We introduce a modified software prediction technique that reduces the IPC loss to 11.4%. This technique is based on a technique used in threaded code interpreters.

A number of hardware mechanisms, including a specialized return address stack and a hardware cache for translated jump target addresses, are studied for efficiently supporting register-indirect jumps. Once all the chaining overheads are removed by these support mechanisms, a superblock-based code cache improves performance due to a better branch prediction rate, improved I-cache locality, and increased chances of straight-line fetches. Simulation results show a 7.7% IPC improvement over a current generation 4-way superscalar processor.

1. Introduction

In recent years a number of systems have been proposed and developed that dynamically translate and/or optimize binary programs from one instruction set to another. Virtually all these systems first map the source binary code into superblocks [19] – code sequences with one entry point and multiple exit points – then translate and/or optimize the superblocks and place them in a code cache for repeated execution on the target platform. When executing within a superblock, performance is enhanced, both because of optimizations that may have been done and because of straight-line instruction fetching that naturally occurs. However, when making transitions from one cached superblock to another, there is a potential for performance loss. For example, if a code cache lookup mechanism must be invoked before each new superblock can be entered, then all performance gains would likely be lost (and then some). One commonly used optimization is to chain superblocks together so that one can immediately branch to the next, but this method only works with direct branches[1]. For indirect jumps, the problem is more difficult and remains a problem in many systems.

In this paper, we study architecture support for efficient control transfers among superblocks being held in a code cache. This support is in the form of a few new instructions and some simple underlying hardware structures. The new instructions could be added to an existing instruction set. (To improve Java performance, Sun Microsystems recently added an instruction that improves indirect jump prediction in their UltraSPARC IIIi processor [32]) Or, if a new proprietary instruction set is the objective, then the architecture support can be included as part of the overall target instruction set architecture (ISA).

1.1 Dynamic code caching

In general, dynamic code caching systems perform basic block re-layout based on observed run-time program characteristics. Hence, a dynamic code caching system can adapt to run-time program behavior changes efficiently. Also dynamic superblocks provide long sequences of instructions that are highly likely to be executed, ideal for most dynamic optimization and high performance binary translation systems.

When program execution begins, the source program binary image is first interpreted. As interpretation proceeds, basic blocks and/or control transfer edges are profiled to find "hotspots", i.e. frequently executed code regions, and/or frequently followed control flow paths.

[1] Following Alpha ISA convention, we use the term *branch* for a control-transfer instruction whose target address is fixed. A register-indirect *jump* finds the target address by reading a specified register.

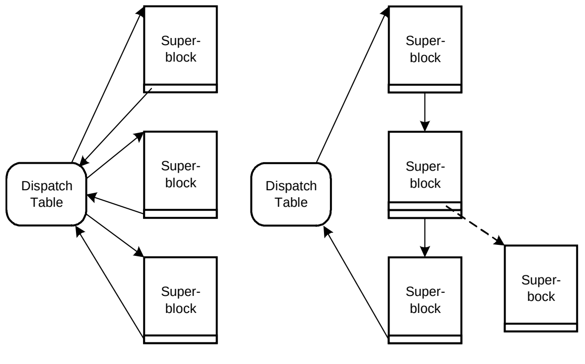
Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36 2003)
0-7695-2043-X/03 $17.00 © 2003 **IEEE**

When the profile data indicates that a certain portion of the program is being frequently executed, then control is passed to a translator (and/or optimizer). The translator forms a superblock, typically using profile data, translates the superblock, and places it in the code cache. It also places an entry in a dispatch table, i.e. a hash table that maps *source* binary *program counter* values (SPCs) to *translated* binary *program counter* values (TPCs). As the program continues, whenever there is a branch or jump in the interpreted code, the hashed SPC is used as an index into the dispatch table. If there is a hit, control is transferred to the mapped superblock in the code cache via the TPC. Similarly, when the end of a superblock is reached, the dispatch table is accessed to find the next superblock (if it exists). At that point if there is a miss in the dispatch table, control is passed back to the interpreter. Initially program execution switches between the interpreter and the code cache frequently, but eventually the program will be executed almost entirely within the code cache.

1.2 Code cache control transfers

In the basic code cache scheme just described there is an obvious steady state performance cost because the dispatch table is accessed every time there is a control transfer (branch or jump) from one superblock to the next. Typically, this would require several instructions, including at least two memory accesses, ending in an indirect jump. Fortunately, direct branches, either conditional or unconditional, are relatively easy to optimize because their (taken) target addresses do not change during program execution. Superblocks can be chained together so that a direct branch from one superblock to another can be made directly without relying on SPC to TPC mapping. Chaining is commonly done in systems that use code caches and is illustrated in Fig. 1.

For indirect transfers, however, the problem is more difficult. Register-indirect jumps have their target addresses stored in a register, and the register value can change over the program's execution. Furthermore, this address is an SPC value, not a TPC value. This means that the original jump target address being held in a register must be translated every time the indirect jump instruction is *executed*, not only when it is *translated*. The most straightforward solution is to consult the dispatch table for every indirect jump. Hence indirect jumps still have a significant performance cost.



(a) without chaining (b) with chaining

Figure 1. Control transfers among superblocks

To save table lookup overhead for each and every indirect jump, many dynamic optimizers/translators implement a form of software-based jump target prediction. In [2][4][5][10][36], a sequence of instructions compares the indirect target SPC held in a register against an embedded translation-time target SPC. A match indicates a correct "prediction" and the inlined direct branch instruction is executed; if not, the code jumps to the shared (slow) dispatch code. Although the example in Fig. 2 shows three sequential predictions, many systems use only one prediction.

```
If Rx == #addr_1 goto #target_1
Else if Rx == #addr_2 goto #target_2
Else if Rx == #addr_3 goto #target_3
Else hash_lookup(Rx); do it the slow way
```

Figure 2. A code sequence that performs indirect jump target comparison

The software prediction method is of limited value, however. First, if the target address is not one of the selected addresses, then time is wasted by testing the possibilities, and the dispatch table lookup has to be performed anyway. The performance cost of a misprediction is high. Second, there are a number of indirect jumps that are not very predictable using this method. For example procedure returns often have a number of call sites, and therefore a number of changing targets. Bruening et al. [4] identify the indirect jump problem as *the* highest overhead in a code cache system and report that a hash table lookup takes 15 instructions, while the software comparison of the target of an indirect jump takes 6 instructions in the x86 ISA.

Previous code cache systems considered the software prediction technique (along with partial inlining of jump target code) as an *optimization*. We find that this technique is rather a *performance limiter*, especially for returns, and we therefore consider alternative methods. We propose an enhanced software prediction technique and study further hardware support mechanisms to achieve zero overhead chaining. We also analyze reasons each method shows different performance characteristics, especially in terms of dynamic instruction count expansion and branch prediction performance changes.

The techniques and support mechanisms studied in the paper can be used to accelerate *any* type of code cache systems – dynamic optimizers, high performance binary translators, or both.

1.3 Related work

There are a variety of systems that use code caching techniques. First, there are transparent optimization systems that do not perform binary translation, but instead focus on dynamic optimizations. These systems include: HP Dynamo [2][4], Wiggins/Redstone [8], Mojo [7], and the system proposed by Merten et al. [27]. Another set of systems rely on code caching as part of a sandboxing framework for program analysis and security enhancement. These systems include DELI [9] and DynamoRIO [5]. As with the dynamic optimizers, these systems do not perform binary translation.

A second important class of systems performs binary translation from one conventional ISA to another (as well as optimization). For example, Strata [30], UQDBT [34] are designed to be retargetable to multiple target platforms. HP Shogun [25] and Aries [37] are used to provide binary compatibility for the existing ISA programs on a platform that executes a new instruction set.

The Transmeta Crusoe Processor/Code Morphing Software [15], IBM BOA [1], and the method proposed by Kim and Smith [24] perform dynamic binary translation from an existing ISA to a proprietary ISA with performance or power efficiency (or both) as a goal. These systems use a code cache to hold superblocks. The IBM DAISY system [10] is similar, except it holds "tree regions" in the cache rather than superblocks.

Finally, high performance high-level language virtual machines (e.g., Java VM) that also use superblock-based code caching technique are emerging [3][35].

With respect to other related systems, there is a clear similarity with hardware trace caches [17][29] (which also cache superblocks, not really "traces" in the Multiflow [26] sense). A hardware trace cache uses hardware rather than software to form superblocks and to manage the trace cache. This difference results in a trade-off: hardware trace caches do not require chaining because the hardware access mechanism is based on source PCs. However, total cache size and maximum allowable superblock size are limited by the amount of on-chip, near-processor storage. It should be noted that hardware trace caches and software code caches are *not* mutually exclusive. For example, many of the above code caching examples [3][4][5][7][34] [35] run on Pentium 4 processor that employs a hardware trace cache [17].

RePLay [12] can be considered an aggressive extension of the hardware trace cache. By converting highly predictable conditional branches into assertions, RePLay increases the average superblock size to the level of software-based code cache systems for more dynamic optimization opportunities.

The system proposed by Merten et al [27] is an innovative hybrid system. Although the code cache is placed in memory, the transition between native execution and "interpretation" is automatically performed by hardware. When an optimized superblock is put into the code cache, its starting TPC is written into the branch target buffer (BTB). The next time a corresponding branch instruction is fetched, control is transferred to the optimized superblock by the BTB. This system also uses the software jump target prediction technique; however when the software prediction fails, program control falls back to the source indirect jump instruction. Therefore no dispatch table lookup is performed. Sooner or later, the program will again reach an optimized superblock. A limitation of this approach is that its application is restricted to dynamic optimization. Many dynamic binary translation systems, such as co-designed virtual machines and high level programming language virtual machines, *cannot* use a hardware interpreter (for executing out-of-code-cache instructions efficiently).

Previously proposed chaining support mechanisms related to our study are explained and compared in their associated subsections.

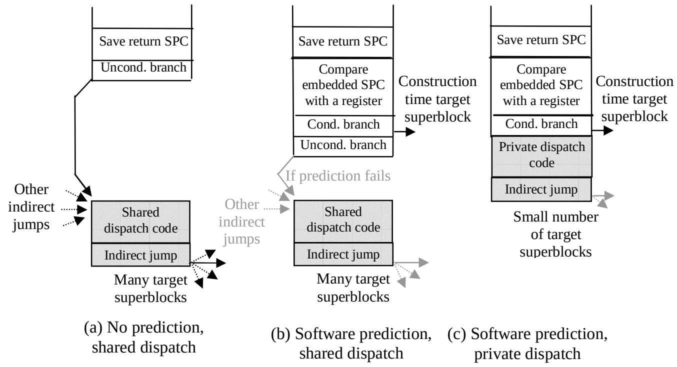
2. Support for code cache control transfers

2.1 Software-based register-indirect jump chaining methods

In Fig. 3, three different software-based indirect jump chaining options are depicted for an indirect function call instruction (e.g. `JSR` in the Alpha ISA)[2]. These software-based methods affect the underlying hardware branch predictor behavior in a negative way as they convert a single indirect jump instruction to a sequence of codes including multiple control transfer instructions. In Fig. 3a, an indirect jump is converted to an unconditional branch to the shared dispatch code. The target address prediction rate of the register-indirect jump in 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 addresses.

The conventional indirect jump chaining method based on software prediction is shown in Fig. 3b. Here, the compare-and-branch code reduces the number of times the shared dispatch code is executed. Hence the pressure on the BTB entry for the indirect jump in the dispatch code is somewhat reduced. However, many times the software prediction is incorrect and in that case two mispredictions can happen – one by the conditional branch in the compare-and-branch code, another by the indirect jump in the

[2] This type of instruction performs two tasks; (1) it saves the next instruction SPC to a register, then (2) jumps to the target SPC stored in a register. The first task is accomplished by a sequence of load-immediate instructions.



(a) No prediction, shared dispatch

(b) Software prediction, shared dispatch

(c) Software prediction, private dispatch

Figure 3. Software-based jump chaining methods

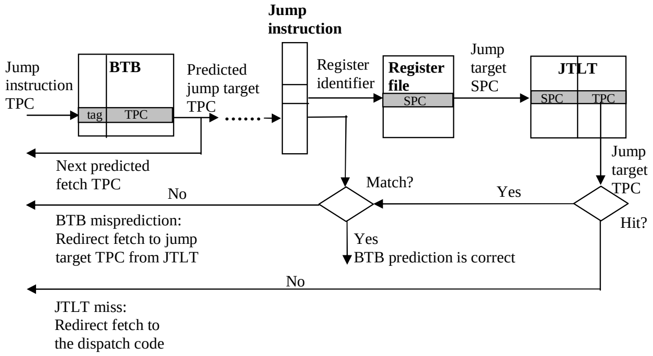


Figure 4. Jump Target-address Lookup Table

dispatch code. The conditional branch has less impact because the branch predictor will eventually be trained to predict the branch as not-taken.

We propose an alternative software method (Fig 3c): replicate the dispatch code after every register-indirect jump, thereby allowing "private" target address prediction in case the superblock construction-time prediction fails. This way the number of mispredictions by the indirect jump in the dispatch code is reduced. This option trades off superblock size, which leads to increased I-cache pressure, for a better target address prediction rate. The private dispatch code concept is similar to the one used in threaded code *interpreters* [11]. However, we are unaware of any previous proposal or existing system that applies this "threaded" technique to a dynamically translated/optimized *code cache* system.

2.2 Jump target-address lookup table

One way to avoid the expensive dispatch table lookup almost entirely is to maintain a hardware cache of dispatch table entries. We call this specialized chaining support feature the Jump Target-address Lookup Table (JTLT). The JTLT is maintained by the code cache manager and always provides a correct translated address if there is a hit. The concept is similar to the software-managed TLBs used in virtual memory systems.

Fig. 4 shows how a JTLT can be used in conjunction with a BTB as a checker/predictor pair. An indirect jump instruction's target TPC is predicted with the normal BTB. This predicted target address flows through the pipeline with the jump instruction itself, just like any other prediction mechanism. When the jump instruction reads the target SPC from its register, the JTLT is searched. If there is a hit at an entry that matches the predicted TPC, the prediction is correct. There are two ways of mispredicting. First, the JTLT itself may miss. In that case, the hardware alone cannot provide the correct target TPC. The jump is not taken and the next sequential instruction, a branch to the dispatch code, is executed. Second, even if a JTLT entry is found, its TPC may be different from the one provided by the BTB. This is a BTB misprediction and fetch is redirected to the TPC from the JTLT.

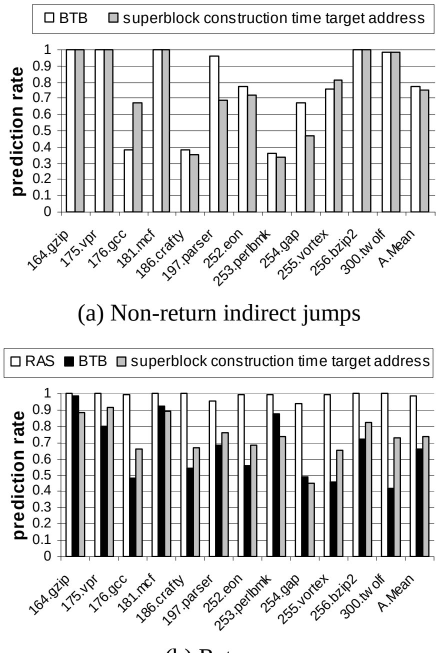
A hardware cache of dispatch table entries and associated instructions have been proposed previously. In [31] a `SEARCH_SWITCH_TABLE` instruction queries the cache with the target SPC in a register; the target TPC is written to another register upon a hit. The next `DYNAMIC_GOTO` instruction reads the latter register and jumps to the TPC. A similar mechanism is proposed in [13]. However, this method is undesirable in systems where no scratch register (for keeping the target TPC) is available to the code cache manager.

It appears that the "Translation Lookaside Buffer" in [22] is also a hardware cache of dispatch table entries. It is not clear, however, exactly how the mechanism is used.

If the JTLT is used, a register-indirect jump is not translated to a compare-and-branch code sequence and remains as an indirect jump. This suppresses dynamic instruction count expansion found in software-based prediction techniques. However, the BTB/JTLT pair also has a couple of weaknesses. First, it requires on-chip storage space. In the Transmeta Crusoe processor a 256-entry "TLB" [15] is used for this purpose. Assuming a 32-bit SPC and a 16-bit TPC, a 256-entry fully-associative JTLT uses 1.5KB of associative memory storage. Second, the BTB/JTLT pair does not provide a highly accurate return address stack (RAS) [21] type prediction capability for return instructions.

2.3 Dual-address return address stack

Most modern processors use a RAS mechanism, which can predict a return instruction's target address very accurately. In a dynamically managed code cache system, however, a conventional RAS cannot be utilized because the saved return target address is a SPC while the corresponding TPC is needed for a return target address prediction. This inability to use a conventional hardware RAS leads to substantial performance loss, as can be seen in Fig. 5. For



(a) Non-return indirect jumps

(b) Returns

Figure 5. Indirect jump target address prediction rates

non-return indirect jumps, the software-based prediction technique is almost as good as the dynamically trained BTB. However, returns are a totally different story. Compared to a RAS, a BTB and the software prediction technique result in 34% and 25% more mispredictions, respectively. Interestingly, the BTB performance is actually *lower* than the quasi-static software prediction

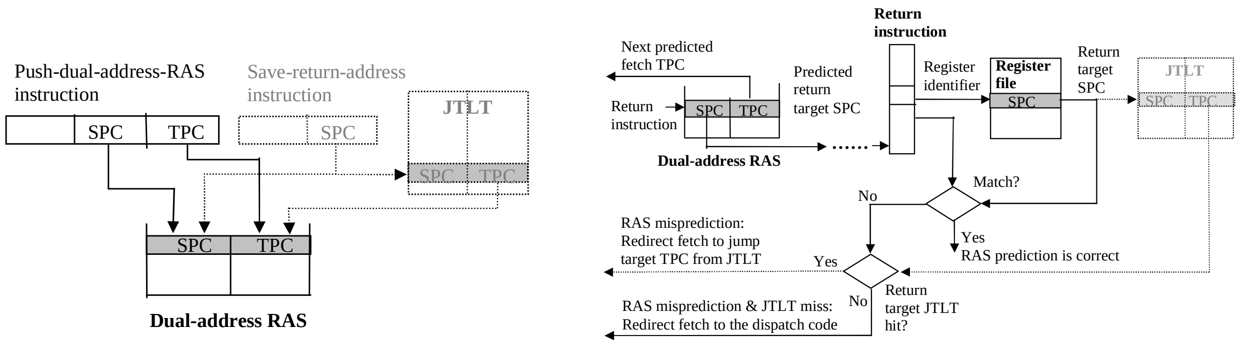
A specialized RAS mechanism that contains an address *pair*, consisting of a return address SPC and its corresponding TPC is shown in Fig. 6. When a return instruction is fetched, the next fetch address is predicted with the popped TPC. The SPC part of the pair flows down the pipeline with the return instruction and is compared to the register value. If the two values do not match, a RAS misprediction is detected and fetch needs to be redirected. If a JTLT is also used, it can be relied upon to provide the correct target TPC. Otherwise, fetch redirection is accomplished by a branch to the dispatch code. Note that the semantics of this return instruction are slightly different from the conventional definition – a conventional return always jumps to the target. Here, if the return prediction is not correct, the next sequential instruction is executed (or the JTLT sets the next TPC if used and is hit).

With this specialized RAS, a return is not converted to the compare-and-branch sequence. The dual address RAS improves the prediction rate for returns and removes many extra instructions that would have been generated for a single return instruction, like the JTLT does.

There are two ways to push a pair of addresses onto the dual address RAS. The first option is to use a special *push-dual-address-RAS* instruction that pushes both return addresses. Finding the return target SPC at superblock construction time is simple. If the corresponding TPC is not found at superblock construction time, an invalid address is written in the TPC field. Later when the return target superblock is constructed, the invalid address is replaced with a valid TPC.

Another way to form a return address pair is to consult the JTLT when the original return SPC is pushed. With the JTLT, the return address TPC does not need to be embedded as in the *push-dual-address-RAS* instruction. When an instruction that saves a return address is encountered, the JTLT is searched for a matching TPC. If a match is found, a pair of return addresses are formed and pushed onto the dual-address RAS. Note that a conventional load-immediate instruction pair that saves a return SPC can not be used in this case. Typically, a load-immediate instruction does not contain a hint to push the RAS. Instead, a special *save-return-address* instruction that contains only the SPC can be used.

In any case, the dual-address RAS should be pushed/popped for all procedure calls and returns – sometimes even with an invalid TPC to maintain correct prediction order.


Push-dual-address-RAS instruction
SPC
TPC
Save-return-address instruction
SPC
JTLT
SPC
TPC
SPC
TPC
Dual-address RAS
Return instruction
Next predicted fetch TPC
Return instruction
SPC
TPC
Dual-address RAS
Predicted return target SPC
Register identifier
Register file
SPC
Return target SPC
JTLT
SPC
TPC
Match?
No
Yes
RAS prediction is correct
RAS misprediction: Redirect fetch to jump target TPC from JTLT
Yes
No
Return target JTLT hit?
RAS misprediction & JTLT miss: Redirect fetch to the dispatch code

(a) RAS push

(b) RAS pop and prediction check

Figure 6. Dual-address return address stack

Table 1. Special instructions to reduce indirect chaining overhead

categories	Instruction	Description
Load long immediate to register	Save-return-address	Save the immediate value (return target SPC) to a register. Also give a hint to the prediction hardware to form a pair of return addresses using JTLT and push it onto the dual-address RAS.
	Push-dual-address-RAS	Contain two immediate values. Save the first immediate value (return target SPC) into a register. Give a hint to the prediction hardware to push both the first and the second (return target TPC) immediate values onto the dual-address RAS.
Conditional indirect jump	Predicted-indirect-jump	Conditionally jump using a register (contains jump target SPC). If there is a JTLT miss, do not jump; the next sequential instruction, an unconditional branch, will branch to the dispatch code.
	Predicted-return	Conditionally jump using a register (contains return target SPC). Give a hint to the prediction hardware to pop the return target TPC from the dual-address RAS. If the RAS prediction is incorrect, either (a) do not jump or (b) jump to the target TPC from the JTLT if the optional JTLT is used.

Table 2. Summary of indirect jump chaining methods

Indirect jump chaining method	Description		
	Dispatch code?	Return prediction mechanism	Related figures
No_pred.no_pred	Always	BTB (an indirect jump in dispatch code)	Fig. 3a
Sw_pred.sw_pred	When SW prediction failed	BTB (a conditional branch plus an indirect jump in dispatch code)	Fig. 3b/c
Sw_pred.ras	When SW prediction failed (non-return jumps) When RAS prediction failed (returns)	RAS	Fig. 3b, Fig. 6
Jtlt.jtlt	When JTLT missed	BTB	Fig. 4
Jtlt.ras	When JTLT missed	RAS	Fig. 4, Fig. 6

The dual-address RAS can be thought as a hardware implementation of the shadow stack mechanism in FX!32 [18]. Similar mechanism was first proposed in [14], then in [24]. In this paper, we provide an alternative return address pair construction method utilizing JTLT.

2.4 Summary of special instructions and indirect jump chaining methods

Table 1 summarizes the special instructions introduced so far. It should be noted that a code cache system can judiciously choose a subset, based on performance requirements, implementation constraints and hardware budget.

Table 2 summarizes the register-indirect chaining methods that we evaluate. Each is named via a pair of terms separated by a period; these are the prediction method for non-return jumps and for returns, respectively. For example, in the *sw_pred.ras* method, software prediction is used for non-return jumps and the dual-address RAS is used for returns.

3. Evaluation methodology

3.1 Isolating code cache effects

We are primarily interested in the analysis and reduction of code cache chaining overhead, whether used in a translation or optimization-only implementation, so we would like to “filter out” the effects of binary optimization and instruction set translation. Therefore, we evaluate the proposed methods by using the “identity translation” where we map the Alpha ISA onto itself and perform no optimizations (other than superblock formation). Of course, the methods proposed and studied in this paper can be applied to systems that dynamically optimize, translate, or both.

It should be noted that a code caching system without any optimization techniques (other than automatic code relayout) is in itself an important design point when strict, bug-for-bug binary compatibility is required.

3.2 Simulation environment

Our aim is to evaluate performance impact of the various chaining mechanisms, including specialized hardware support. To do this, we have written a code caching system in C and integrated it with a timing simulator. The simulated code cache system performs all interpretation and superblock construction in the same sequence as in actual implementation. The timing simulator part of the simulation system is based on the SimpleScalar 3.0C toolset [6] and is heavily modified to closely model modern processor pipeline designs. The same timing simulator without the code caching facility is also used as a baseline for comparisons. This baseline configuration is referred to as *original*.

When program control flow reaches an existing superblock in the code cache, the simulator begins detailed timing simulation. Here the timing simulation starts with an initially empty pipeline. Similarly if an exit condition from the code cache is met (i.e., a superblock exit instruction's target is not currently cached in the code cache), the mode is changed to interpretation after the last instruction in the pipeline is committed. Overall performance is then measured as source instructions per cycle (IPC) for execution of all cached (and chained) instructions.

To collect statistics, we use the SPEC CPU2000 integer benchmarks compiled for the Alpha EV6 ISA at the `base` optimization level (`–arch ev6 –non_shared –fast`). Note that the `–fast` option includes aggressive instruction scheduling, procedure inlining, and loop unrolling. The compiler flags are the same as those reported for Compaq AlphaServer ES40 SPEC CPU2000 submission 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 to completion or 4.3 billion instructions.

NOPs defined in the Alpha ISA are properly recognized and removed when superblocks are formed. NOPs are usually generated by the compiler to align control-transfer target addresses to I-cache line boundaries. In *original*, NOPs are removed by the hardware in the decoding stage. We show the performance impact of removing NOPs in superblocks in section 4.3.

Our superblock formation algorithm is a slightly modified version of Dynamo's Most Recently Executed Tail (MRET) heuristic [2]. Unlike Dynamo, we stop constructing a superblock when an indirect jump is encountered. We use a maximum superblock size of 200 instructions and a tail execution counter threshold of 50.

3.3 Simulated processor pipeline

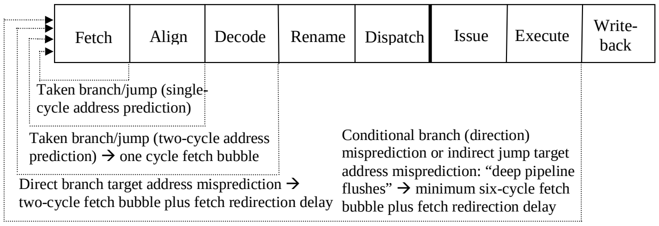

Fetch
Align
Decode
Rename
Dispatch
Issue
Execute
Write-back
Taken branch/jump (single-cycle address prediction)
Taken branch/jump (two-cycle address prediction) → one cycle fetch bubble
Direct branch target address misprediction → two-cycle fetch bubble plus fetch redirection delay
Conditional branch (direction) misprediction or indirect jump target address misprediction: "deep pipeline flushes" → minimum six-cycle fetch bubble plus fetch redirection delay

Figure 7. Simulated pipeline showing misprediction penalties

For the purposes of control transfers, an eight-stage out-of-order superscalar processor pipeline simulator is used (additional pipeline stages for the non-control transfer instructions are not shown). The pipeline is shown in Fig. 7. Instruction fetch takes two cycles – one cycle for accessing the I-cache SRAM array and another cycle for shift and mask operation to select valid fetch target instructions. Control transfer information such as branch/jump types and their locations within an I-cache line is pre-decoded and stored in the I-cache array when the cache line is

Table 3. Simulated microarchitecture parameters

	4-way issue microarchitecture	8-way issue microarchitecture
Branch prediction	16K-entry, 12-bit global history g-share branch predictor; 16-entry RAS; 2K-entry, 4-way set associative BTB; 256-entry fully associative, LRU-replacement JTLT (if used)	
Branch predictor bandwidth	Up to 1 prediction per cycle	Up to 2 predictions per cycle
L1 I-cache	32-KB size, direct-mapped, 2-cycle hit latency	
	64-byte line size	128-byte line size
Fetch bandwidth	Maximum 4 instructions per cycle; Fetch continues after the first predicted not-taken conditional branch; Fetch stops if a second branch is found	Maximum 8 instructions per cycle; Fetch continues after the first and second predicted not-taken conditional branches; Fetch stops if a third branch is found
L1 D-cache	32-KB size, 4-way set-associative, random replacement, 64-byte line size, 2-cycle hit latency	
Unified L2 cache	1-MB size, 4-way set-associative, random replacement, 128-byte line size, 8-cycle hit latency	
Memory	128-cycle latency, 64-bit wide, 4-cycle burst	
Decode/issue/ retire bandwidth	4	8
Issue window size	64	128
Execution resources	4 integer units, 2 L1 D-cache ports, 2 floating-point adders, 2 floating-point multipliers	8 integer units, 4 L1 D-cache ports, 4 floating-point adders, 4 floating-point multipliers
Reorder buffer size	128	256

brought into the I-cache. All branch prediction mechanisms, i.e., branch (direction) predictor, BTB, and RAS can be configured as either single-cycle or two-cycle. This is to model the effect of multi-cycle branch prediction mechanisms [20] found in current pipelined processors. For example, the IBM POWER4 [33] and AMD Opteron [23] have such multi-cycle predictors which create fetch bubble(s) even for correctly predicted taken branches. Regarding predicted directions of conditional branches, when there is a disagreement between branch predictor and BTB for a conditional branch, the branch prediction overrides the BTB prediction.

For machine resources, we use two sets of microarchitecture parameters (Table 3). The first set models a moderate-width processor, similar to current generation designs. The second set models a relatively aggressive processor. All evaluations are done with the 4-way issue microarchitecture, except where noted.

4. Performance results

We first consider characteristics of cached code (Table 4). From the 3rd column of the table, it is apparent that all benchmark programs almost always execute within the code cache – even for these very short (in real terms) benchmark runs. On the other hand, the total static code cache size in the 5th column indicates that the code working set sizes of the benchmark programs, with possible exception of *176.gcc*, are fairly small. Of primary interest in this data is the average number of instructions between taken control transfer instructions in the last column. On average, dynamic superblock caching achieves about a two-fold increase compared to original program execution.

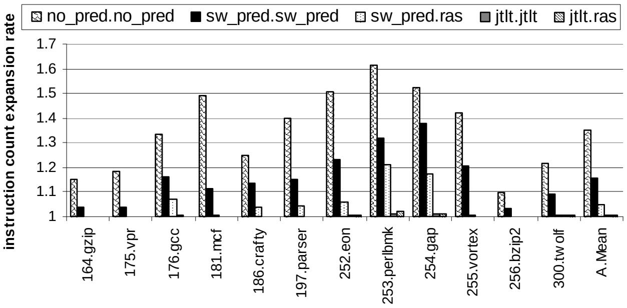


Figure 8. Dynamic instruction count expansion

Another important statistic is the number of extra instructions generated by indirect chaining methods. Fig. 8 shows the dynamic instruction count expansion rates when the dispatch code consumes 20 instructions. It is obvious that without any register-indirect jump chaining support, as in the *no_pred.no_pred* method, program performance will be unacceptable as 35% more instructions have to be executed. Conventional software prediction in *sw_pred.sw_pred* method cuts the number to about 16% by executing only the relatively short compare-and-branch code when the prediction is correct (Both shared and threaded versions result in the same instruction count expansion). Providing a dual-address RAS reduces another 10.6% of the total instructions. This is not only because return instructions now seldom reach the dispatch code, but also because the compare-and-branch code is not generated for a source return instruction. Similarly, JTLT removes almost all extra instructions for all jumps.

Table 4. General superblock characteristics

Benchmark	No. of dynamic source instructions	% of instructions executed in code cache	No. of static superblocks	Total cached code size (bytes)	Superblock completion rate	Average number of instructions between taken control transfer instructions	
						original	code cache
164.gzip	3.50 billion	0.9999	366	53,188	0.76	13.6	27.3
175.vpr	1.54 billion	0.9996	467	74,284	0.58	13.7	28.2
176.gcc	1.89 billion	0.9938	11,599	1,877,468	0.73	9.7	19.1
181.mcf	259 million	0.9989	214	23,472	0.70	8.7	10.1
186.crafty	4.18 billion	0.9995	1,665	336,164	0.55	12.7	30.0
197.parser	4.07 billion	0.9996	2,441	335,948	0.77	8.1	13.9
252.eon	95 million	0.9899	633	79,756	0.88	14.3	23.6
253.perlbmk	4.29 billion	0.9998	426	356,164	0.91	10.4	18.8
254.gap	1.2 billion	0.9978	2,630	411,768	0.80	9.9	19.2
255.vortex	4.29 billion	0.9991	2,547	707,212	0.91	10.7	36.3
256.bzip2	4.29 billion	0.9999	235	27,100	0.96	14.0	20.1
300.twolf	253 million	0.9946	873	127,940	0.61	14.5	23.2
Average		**0.9977**			**0.76**	**11.7**	**22.5**

4.1 Branch prediction performance

Chaining has a significant effect on a program's branch prediction characteristics because it can add extra control-transfer instructions or can even remove some source control transfer instructions (i.e., unconditional direct branches inside a superblock). For direct conditional branches, chaining does not change prediction performance significantly. Nonetheless, a lower number of taken branches and inlined unconditional branches tend to reduce pressure on the branch prediction hardware. On the other hand, chaining of register-indirect jumps does have a large effect on branch prediction performance, and each scheme exhibits different branch prediction characteristics. Fig. 9 shows detailed breakdown of all control transfer mispredictions that are resolved after the instruction is executed.

First, note that the performance impact of register-indirect jump mispredictions was not very significant in the *original* program execution. However their effect is exacerbated in a code cache system. The conventional chaining method, *sw_pred.sw_pred*, experiences 46% more mispredictions than *original*. This increase is mostly due to the mispredictions of the indirect jump in the shared dispatch code. A threaded version, *sw_pred.sw_pred (threaded)* reduces this type of mispredictions by 44% thanks to the private dispatch code. However it still generates 23% more mispredictions than *original*. Introducing the dual-address RAS further reduces the indirect jump misprediction to the level of *original*.

The JTLT also reduces mispredictions by cutting the dispatch code execution frequency. However, *jtlt.jtlt* still has 24% more mispredictions than the original program execution. This is about the same as the best software-based method, *sw_pred.sw_pred (threaded)*. This may seem surprising at first but it does make sense considering the prediction performance in Fig. 5.

The best method, *jtlt.ras*, has 5.6% fewer overall mispredictions than *original* due to a reduction in conditional branch mispredictions. This is possible because fewer taken branches reduce negative interference in the branch predictor pattern history table [28].

It should be pointed out that branch prediction performance comes close to the original program only after introducing the dual-address RAS. Interestingly, *sw_pred.ras* produces *fewer* mispredictions than *jtlt.jtlt*, the hardware-intensive technique.

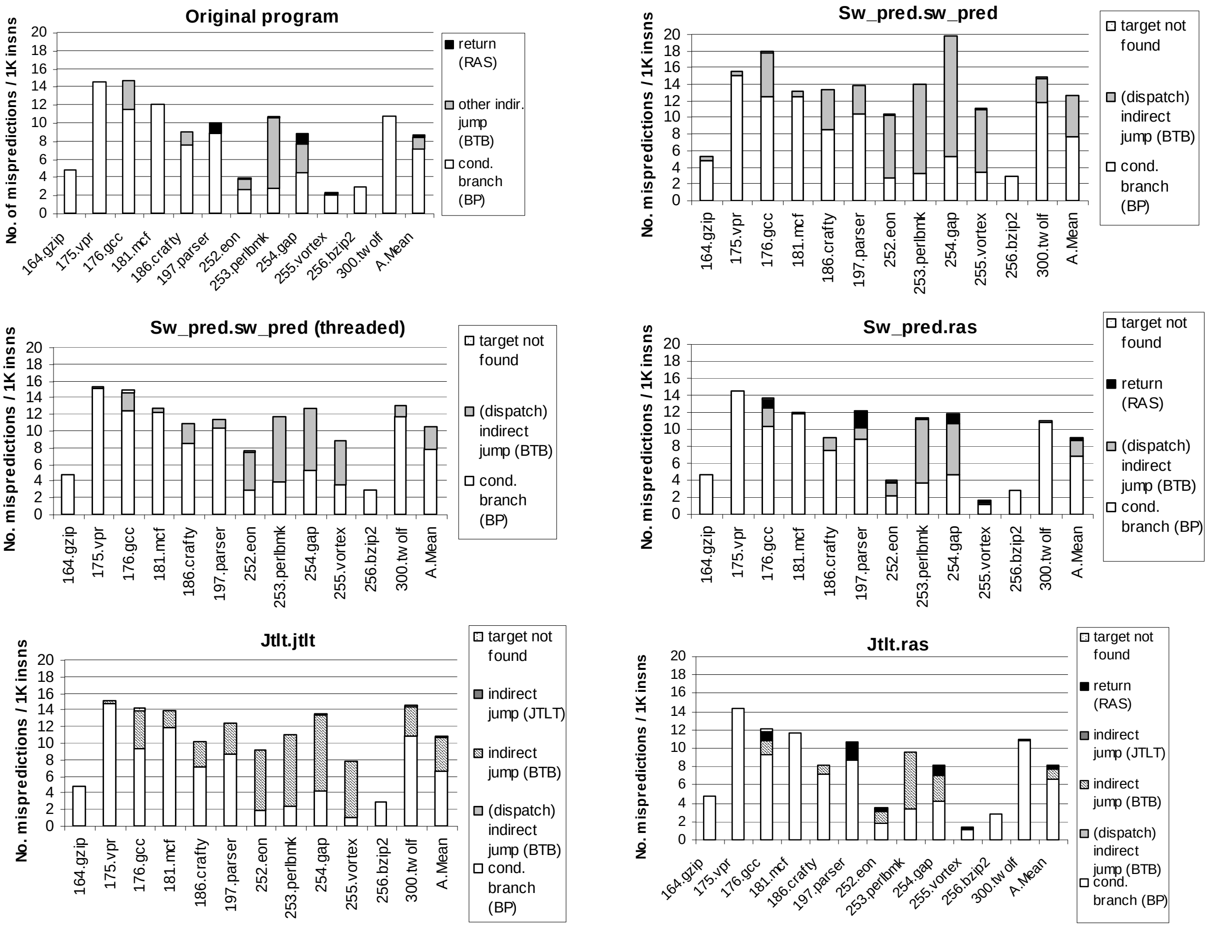


Figure 9. Classification of control transfer mispredictions

4.2 I-cache performance

Another important program characteristic that can be affected by the chaining method is I-cache performance. Fig. 10 shows that, in general, superblock-based code caching helps reduce I-cache misses, except for the threaded variation which suffers more I-cache misses due to the replicated dispatch code. That is, improved I-cache locality by superblock caching works to offset increased I-cache pressure from chaining (as is implied by the dynamic instruction count increase).

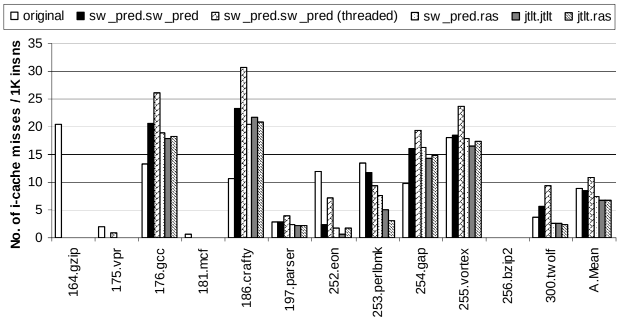


Figure 10. Number of I-cache misses

Of special interest is the dramatic miss reduction in *164.gzip*. Here, code re-layout eliminates cache thrashing. Although this is a valid optimization, it is probably limited to direct-mapped caches. If *164.gzip* is omitted, the best methods that use JTLT show a 6.3% I-cache miss reduction. (24.3% if *164.gzip* is included).

4.3 IPC performance

Fig. 11 shows overall performance in terms of the original source IPC. The results show that the conventional indirect jump method that relies on software prediction (*sw_pred.sw_pred*) performs poorly, resulting in 14.6% IPC *loss*. Here, improved fetch bandwidth is offset by the chaining overhead, mostly due to increased branch mispredictions and extra instructions. Interestingly, this result contradicts previous results reported in [2] where a 6% *speedup* is reported just through superblock code caching. We believe the following factors contribute to the difference in results:

- Efficiency of hardware prediction mechanisms: the PA-8000 processor, used in [2], does not predict indirect jumps and always stalls fetch until the target address is resolved [16]. Hence, converting a register-indirect jump to the software prediction compare-and-branch code greatly reduces fetch stall cycles in the PA-8000. In contrast, our simulation model predicts jump target with a BTB and does not stall fetch, so a similar benefit is not realized. This is confirmed by [4]: where the Dynamo system was ported to a Pentium II platform (which does predict indirect jump targets with a BTB), resulting in substantial slowdowns due to indirect jumps – even worse than we report here.
- Differences in the superblock formation algorithm: we stop constructing a superblock whenever an indirect jump is encountered. Hence some straight-line fetch optimization opportunities are not exploited.

Returning to Fig. 11, a threaded version, *sw_pred.sw_pred (threaded)*, performs 3.2% better than the conventional *sw_pred.sw_pred*, showing that indirect jump prediction performance improvements more than offset any losses in I-cache performance when replicated dispatch code is used. Even this best performing software-only method still lags original program performance by 11.4%.

It is only after specialized hardware mechanisms are introduced that the identity-translation code cache system outperforms original program execution. Referring to *jtlt.jtlt*, the introduction of the JTLT greatly enhances performance both by suppressing extra instructions and in improving predictor performance. As a result, *jtlt.jtlt* achieves a 4.6% performance improvement over *original*.

However, even more important is the effect of the dual-address RAS. This is evident in the *sw_pred.ras* method; 2.1% IPC improvement is achieved *without* requiring any extra on-chip storage as in the JTLT. This is a 15.4% improvement over the best performing software-only method (and a 19.6% improvement over the conventional method, *sw_pred.sw_pred*). Finally, combining both the JTLT and

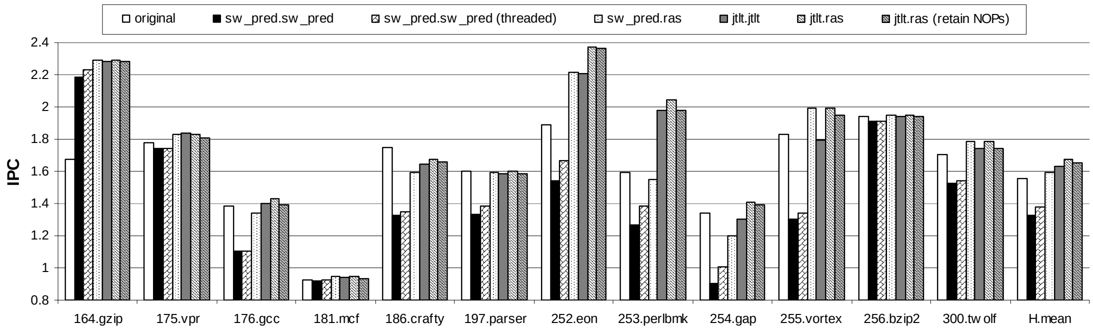
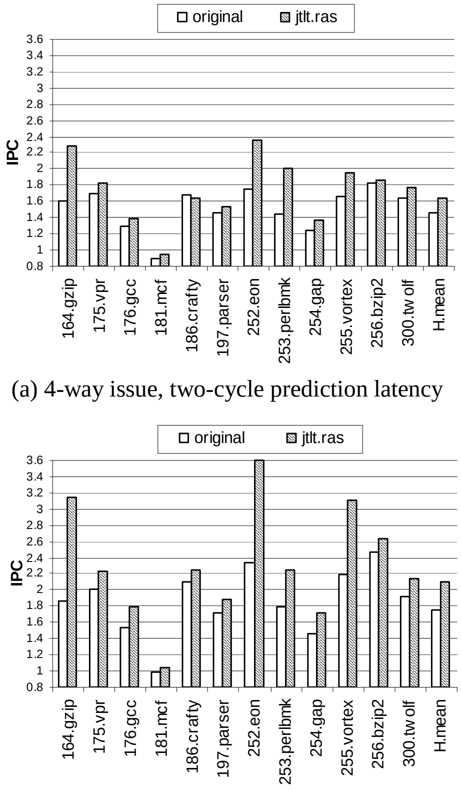


Figure 11. IPC comparisons

dual-address RAS (*jtlt.ras*) results in a 7.7% IPC improvement over *original*.

Next we consider the effect of Alpha NOP removal in superblocks. The benefit can be seen by looking at the performance of *jtlt.ras (retain NOPs)*. If not removed, these NOPs put more pressure than necessary on the fetch mechanism and reduce effective fetch bandwidth. The results show that 1.5% of the total 7.7% IPC improvement comes from NOP removal in superblocks.



(a) 4-way issue, two-cycle prediction latency

(b) 8-way issue, two-cycle prediction latency

Figure 12. Effects of machine parameters

Fig. 12a shows the IPC performance improvement (*jtlt.ras* IPC over *original* program IPC) when the effect of two-cycle predictors is accounted for. Here, both suffer from fetch bubbles generated by taken branches and jumps. However, the reduced number of taken branches in *jtlt.ras* allows it to tolerate the predictor latencies better. On average, *jtlt.ras* performs better than *original* by 12.8%.

Finally the effect of larger maximum fetch bandwidth that would be necessary to support a rather aggressive 8-way issue processor is shown in Fig. 12b. Here, predictor latencies are set at two-cycles. The results show that superblock caching scales better with increased I-cache line size and branch prediction bandwidth. On average, relative IPC improvement of *jtlt.ras* over *original* is 18.8%.

5. Conclusions

In this paper, we studied in detail variety of code cache chaining techniques including several specialized architecture support mechanisms. We identify the lack of accurate return prediction as the biggest performance limiter in code caching systems. We showed that the dual-address return address stack is a cost-effective solution to enhance the performance of a code caching system. The jump target-address lookup table – a hardware cache of the dispatch table – also helps to further reduce the chaining overhead. Other aspects of chaining were also enhanced; a dynamic threaded code technique was applied to improve the software-based jump prediction method.

In summary, the techniques studied in this paper can be used in many code caching systems. For co-designed virtual machine systems, the full set of specialized hardware mechanisms can be used. Other systems can judiciously select the most cost-effective mechanisms. Even the strictly software-based code cache systems can benefit from the dynamic threaded code technique.

As the importance of dynamic optimization and binary translation grows, we believe the mechanisms studied in this paper will provide a high performance base framework to develop further optimizations.

6. Acknowledgements

This work is being supported by SRC grant 2001-HJ-902, NSF grants EIA-0071924 and CCR-0311361, 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 RC21665*, Dec. 2000.

[2] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, "Dynamo: A Transparent Dynamic Optimization System," *Conf. Programming Language Design and Implementation*, pp. 1-12, Jun. 2000.

[3] Marc Berndl, Laurie Hendren, "Dynamic Profiling and Trace Cache Generation," *Int. Symp. Code Generation and Optimization*, pp. 276-285, Mar. 2003.

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

[5] Derek Bruening, Timothy Garnett, Saman Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization," *Int. Symp. Code Generation and Optimization*, pp. 265-275, Mar. 2003.

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

[7] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, David M. Gillies, “Mojo: A Dynamic Optimization System,” *The 3rd Workshop on Feedback-Directed and Dynamic Optimization*, Dec. 2000.

[8] Dean Deaver, Rick Gorton, Norman Rubin, “Wiggins/Redstone: An Online Program Specializer,” *The 11th HotChips Symposim*, Jun. 1999.

[9] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, Joseph A. Fisher, “DELI: A New Run-Time Control Point,” *The 35th Int. Symp. Microarchitecture*, pp. 257-268, Dec. 2002.

[10] Kemal Ebcioglu, Erik R. Altman, Michael Gschwind, Sumedh Sathaye, “Dynamic Binary Translation and Optimization,” *IEEE Trans. Computers*, Vol. 50, No. 6, pp. 529-548, Jun. 2001.

[11] M. Anton Ertl, David Gregg, “The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures,” *Europ. Conf. Parallel Computing*, pp. 403-412, Aug. 2001.

[12] 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,” *The 34th Int. Symp. Microarchitecture*, pp. 16-27, Dec. 2001.

[13] Michael Gschwind, “Method and Apparatus for Determining Branch Addresses in Programs Generated by Binary Translation, *IBM Disclosures YOR819980334*, Jul. 1998.

[14] Michael Gschwind, “Method and Apparatus for Rapid Return Address Computation in Binary Translation,” *IBM Disclosures YOR819980410*, Sep. 1998.

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

[16] Hewlett Packard Co., “PA-RISC 8x00 Family of Microprocessors with Focus on PA-8700,” www.cpus.hp.com/technical_references/PA-8700wp.pdf.

[17] 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.

[18] Raymond J. Hookway, Mark A. Herdeg, “Digital FX!32: Combining Emulation and Binary Translation,” *Digital Technical Journal*, Vol. 9, No. 1, pp. 3-12, Jan. 1997.

[19] 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, Daniel M. Lavery, “The Superblock: An Effective Technique for VLIW and Superscalar Compilation,” *The Journal of Supercomputing*, Kluwer Academic Publishing, pp. 229-248, 1993.

[20] Daniel A. Jimenez, Stephen W. Keckler, Calvin Lin, “The Impact of Delay on the Design of Branch Predictors,” *The 33rd Int. Symp. Microarchitecture*, pp. 67-76, Dec. 2000.

[21] David Kaeli, P. G. Emma, “Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns,” *The 18th Int. Symp. Computer Architecture*, pp. 34-42, Jun. 1991.

[22] Edmund J. Kelly, Robert F. Cmelik, Malcolm J. Wing, “Memory Controller for a Microprocessor for Detecting a Failure of Speculation on the Physical Nature of a Component Being Addressed,” *US Patent 5,832,205*, Nov. 1998.

[23] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, Pat Conway, “The AMD Opteron Processor for Multiprocessor Servers,” *IEEE Micro*, Vol. 23, No. 2, pp. 66-76, Mar/Apr 2003.

[24] Ho-Seop Kim, James E. Smith, “Dynamic Binary Translation for Accumulator-Oriented Architectures,” *Int. Symp. Code Generation and Optimization*, pp. 25-35, Mar. 2003.

[25] Bich C. Le, “An Out-of-Order Execution Technique for Runtime Binary Translators,” *The 8th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 151-158, Oct. 1998.

[26] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O’Donnell, John C. Ruttenberg, “The Multiflow Trace Scheduling Compiler,” *The Journal of Supercomputing*, Kluwer Academic Publishing, pp.51-142, 1993.

[27] Matthew C. Merten, Andrew R. Trick, Erik M. Nystrom, Ronald D. Barnes, Wen-mei W. Hwu, “A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hotspots,” *The 27th Int. Symp. Computer Architecture*, pp. 59-70, Jun. 2000.

[28] Alex Ramirez, Josep L. Larriba-Pey, Matero Valero, “The Effect of Code Reordering on Branch Prediction,” *The 9th Int. Conf. Parallel Architectures and Compilation Techniques*, pp. 189-198, Oct. 2000.

[29] Eric Rotenberg, Steve Bennett, James E. Smith, “Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching,” *The 29th Int. Symp. Microarchitecture*, pp.24-34, Dec 1996.

[30] Kevin Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, M L. Soffa, *Int. Symp. Code Generation and Optimization*, “Retargetable and Reconfigurable Software Dynamic Translation,” pp. 36-47, Mar. 2003.

[31] Gabriel M. Silberman, Kemal Ebcioglu, “An Architectural Framework for Supporting Heterogeneous Instruction-Set Architectures,” *IEEE Computer*, Vol. 26, No. 6, pp. 39-56, 1993.

[32] Sun Microsystems, “UltraSPARC IIIi Processor,” www.sun.com/processors/UltraSPARC-IIIi/us3i_datasheet.pdf, 2003.

[33] Joel M. Tendler, Steve Dodson, Steve Fields, Hung Le, Balaram Sinharoy, “POWER4 System Microarchitecture,” *IBM Journal of Research and Development*, Vol. 46, No. 1, pp. 5-26, Jan. 2002.

[34] David Ung, Cristina Cifuentes, “Optimizing Hot Paths in a Dynamic Binary Translator,” *The 2nd Workshop on Binary Translation*, Oct. 2000.

[35] John Whaley, “Partial Method Compilation using Dynamic Profile Information,” *Int. Conf. Object-Oriented Programming, Systems, Languages & Applications*, pp.166-179, 2001.

[36] Emmett Witchel, Mendel Rosenblum, “Embra: Fast and Flexible Machine Simulation,” *The Conf. Measurement and Modeling of Computer Systems*, pp. 68-78, May 1996.

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